Parallel Computational Complexity and Date-Transfer Complexity of Supercomputing(Dissertation_全文)

AUTHOR(S):

Okabe, Yasuo

Parallel Computational Complexity and Data-Transfer Complexity of Supercomputing

January 1994

Yasuo OKABE

Parallel Computational Complexity and Data-Transfer Complexity of Supercomputing

January 1994

Yasuo OKABE

Abstract

With the recent advance in hardware technology of very large-scale integration and parallel computer architecture, vector and parallel *supercomputers* have become widely used in various fields of science. Two most attracted issues in fundamental software science of supercomputing are, design and analysis of efficient parallel algorithms, and development of powerful system softwares. This thesis discusses the both with respect to parallel computational complexity and data-transfer complexity.

In Chapter II, parallel computational complexity of logic programs is discussed. Shapiro first showed that there is a close relationship between logic programs and alternating Turing machines. Here more precise correspondence between complexity of logic programs and that on conventional models, especially circuit complexity, are shown. Well-known complexity classes such as $\mathcal{NC}$, $\mathcal{P}$, $\mathcal{NP}$, etc. are characterized via logic programs.

In Chapter III, parallelization of logical query programs are discussed. Logical query programs, i.e., logic programs without function symbols, are used as recursive inference rules of deductive databases. Parallel computational complexity of logical queries was investigated Ullman and Van Gelder. Here a subclass of logical query programs called *non-confluent programs* are defined, and it is shown that the class of queries defined by non-confluent programs is exactly equal to LOGCFL (and hence in $\mathcal{NC}$). A procedure of program transformation which parallelize an ordinary non-confluent program into one with log depth complexity is also presented.

In Chapter IV, a new parallel computation model of two-level storage is proposed. The model is composed of processors with internal memory and a disk as a file server, which are connected by a shared bus. Computation time is measured by the number of data-transfer operations via the bus. The power of the model es-

sentially depends on whether *broadcasting* to the bus is allowed or not. Here tight lower and upper bounds of the number of required data transfer operations for sorting, permutation, FFT, etc., on both of models with and without broadcasting. The results are multiprocessor extensions of the bounds on the uniprocessor two-level memory model shown by Aggarwal and Vitter.

In Chapter V, virtualization of hierarchical memory of vector supercomputers is discussed. Most vector supercomputers are equipped with high-speed semiconductor secondary storage, called extended storage. The well-known virtual storage management technique commonly used on general-purpose computers, however, would tragically decline the performance of large-scale numerical computing. The approach of this thesis is transforming a program written without considering the presence of the memory hierarchy automatically so that it does appropriate data transfers between main memory and extended storage. Algorithms for extracting declarations of huge arrays and references to them from an ordinary program, partitioning the arrays and allocating them on extended storage, and inserting codes which control data transfers between extended storage and temporary area on main memory are presented. Vectorizability of the program is preserved by the transformation by utilizing lattice-wise partitioning of huge arrays into pages, page prefetching before vector-mode instructions, and indirect vector load/store of pages randomly scattered on the temporary area. The transformed code of an LU-factorization program achieves about 50% of the execution speed when the original one is executed normally on main memory.

Contents

List of Figures

List of Tables

CHAPTER I

Introduction

1. Background

With the recent advance in hardware technology of very large-scale integration and parallel computer architecture, vector and parallel *supercomputers* have become widely used in various fields of science [Fer86]. The peak performance of them have been growing about ten times or more every five years [HJ88a]. Massively parallel supercomputers are also used as special-purpose hardware for hard-nut problems which would need too much time on vector or mediocre-parallel computers [Tom86a]. Two most attracted issues in fundamental software science of supercomputing are, design and analysis of efficient parallel algorithms, and development of powerful system softwares.

Needless to say, the aim of parallel computing is to solve problems as fast as possible. In the theory of computational complexity, algorithms for a problem are evaluated and compared on a specific hardware computation model so as to derive the theoretically minimum computation time required to solve the problem on it [Yas92, Iwa92a]. Typical parallel computation models commonly used are, parallel random access machines (PRAMs) [FW78, SV84], alternating Turing machines [CKS81, Ruz80], etc. In particular, the combinational circuit model [Sha49, Ruz81] is the most powerful, stable, and realistic theoretical model of parallel computing

hardware, since it is based on physically realizable special-purpose circuits with as much parallelism as possible.

Among theoretical models of parallel computing software, logic programming languages have lately attracted considerable attention with the researches on "fifth generation computer systems." Logic programming languages are based on first-order predicate logic and are easily treated in mathematical formalization, but are different from conventional procedural programming languages in many respects. The first topic of this thesis is analyzing parallel computational complexity of logic programs. A mathematical foundation of parallelism of logic programs in terms of computational complexity theory was given first by Shapiro [Sha84]. He showed that there is a close relationship between logic programs and alternating Turing machines (ATMs). More precise correspondence between complexity of logic programs and that on conventional models, especially circuit complexity, is shown in Chapter II.

Another mathematical formalization of logic programming is found in the theory of deductive databases. Relational calculus based on first-order predicate logic is simple and efficiently parallelizable, but not a few practically important queries cannot be written in first-order relational calculus (e.g., ancestor-descendant relation cannot derived from parent-child relation). Deductive database is an extension of relational database so as to allow recursive operation, such as transitive closure or the least fixed point operator. Logical query programs (or referred as Datalog programs) are logic programs without function symbols, and can be regard as first-order query minus negation plus the least fixed point operation. Any logical query is computable in deterministic polynomial time of the size of database [CH84], but it may still cost unrealisticly much for very large databases. Ullman and Van Gelder [UV88] investigated $\mathcal{P}$-complete queries and queries computable in polylogarithmic time by PRAMs with polynomial number of processors, viz.

queries in $\mathcal{NC}$. The class $\mathcal{NC}$ is the subclass of $\mathcal{P}$ (PTIME) consists of all problems computable in high speed by parallelization [Coo85], and it has been zealously studied for the recent ten years [Miy91, Iwa92b]. The second topic of this thesis is syntactic characterization of "effectively parallelizable" logic query programs. A concrete procedure for mechanical parallelization of queries in $\mathcal{NC}$ is presented. Known optimizing procedures for logical query programs mostly target removing redundancy in recursive rules [Sag88, Nau89, GMS93]; these effectively reduce the computation time under OR-parallelism, but not necessarily reduce the computation time under AND/OR-parallelism. The procedure shown in Chapter III is distinctive in gaining fully parallelism by introducing some syntactically redundant rules.

Parallel computing in the real world cannot always be performed with so much parallelism as in the theories of parallel computation mentioned above. Actual parallel computers have limitations in the number of processors, the size of main memory, and the memory-bus architecture. We can hardly prepare $N^{O(1)}$ processors for a given problem of size N, when N is as large as worth supercomputing! Thus the number of available processors, say p, must be evaluated as a parameter independent of N. While a PRAM has shared memory read and written concurrently by all processors, actual parallel computer architectures allow at most twenty processors to share one memory bus. Massively parallel processors in the real world have local memory each and are connected with a certain network topology. Computation time on such machines is dominated by the number of communications among processors. A problem instance of supercomputing itself often overflows the size of internal (either shared or distributed) memory. In such case, computation time is mostly dominated by the number of required I/Os between main memory and secondary storage.

The number of required I/Os in computing a large-size problem instance which

does not fit within the internal memory is called *I/O complexity*. Floyd [Flo72] investigated I/O complexity of permuting under two-level memory hierarchy. His computation model is composed of a processor with internal memory and a disk, and data are transferred in blocks between the two. Here parallelism resides in data transfers: the parallelism in a block data transfer. An extension of Floyd model is a processor with multiple disks [AV88, VS90]; another dimension of parallelism resides in parallel disk I/Os. On the other hand, parallel algorithms on highly parallel computers with various network topologies have been studied by many researchers [Aki85, Ume90, Ume91]. Recently mesh-connected computers with broadcast buses are particularly studied because of the suitability of VLSI implementation. *Broadcasting* is the third dimension of parallelism resides in data transfers. To the best of the author's knowledge, however, there have been no published results concerning *block data-transfer* complexity on a specific connection network together with block I/O complexity under memory hierarchy. The third topic of this thesis is analyzing the data-transfer complexity of some problems on a realistic parallel two-level memory architecture with a broadcasting bus.

The forth topic of this thesis is memory hierarchy on vector supercomputers. Since the memory demand of large-scale supercomputing is far beyond the largest possible size of main memory, most vector supercomputers are equipped with semiconductor secondary memory device called extended storage [ONK86, KaHMK+88, WKI86]. Data transfer between main memory and extended storage is done in high-speed block transfer mode; thus data on extended storage cannot be accessed in the same way as those on main memory. It is shown that almost the same performance can be achieved allocating huge array data on extended stotage as when all data are on main memory, by designing optimal vector algorithms which require minimum data transfers [TO87, TS88]. But re-designing algorithms and re-coding whole of a program would be too much toil for ordinary users. How-

ever, the well-known virtual storage management technique, which is commonly used on general-purpose computers, sometimes tragically declines the performance of large-scale numerical computing; this problem is known as working-set anomaly [ASP82]. The approach of this thesis is distinguished from existent works, in transforming a program written without considering the presence of the memory hierarchy automatically so that it does appropriate data transfers between main memory and extended storage. Optimizing data transfers is pursued as far as the vector algorithm for arithmetic operations in the original program is completely preserved.

2. Outline of the thesis

This thesis discusses theoretical and fundamental aspects of supercomputing softwares. The four topics stated in the previous section are described in the following four chapters respectively.

The first and the second topics are concerned with computational complexity of logic programs, and are argued from an "extremist-in-parallelism" viewpoint [Iwa90]. Parallel complexity and parallelizability is evaluated with computation time of on-processor operations on any number of processors required (although the number of processors is limited to a polynomial of the size of the problem instance), while the time required for inter-processor communication is ignored. In contrast, the third and forth topics are concerned with data-transfer complexity and optimal data-transfer algorithms. In the meanwhile, the first and the third topics are concerned with the lower bound of the computation cost of a problem and the optimal algorithm for it, while the second and the forth topics are concerned with system softwares which fully optimize parallel programs written by ordinary users.

In Chapter II, we improve Shapiro's results on relations between logic programs and alternating Turing machines, to cope with alternating sublinear time and space

complexity. In particular, a close relationship between logic programs and tree-size bounded alternating Turing machines is shown. Applying the correspondence of uniform combinational circuits and alternating Turing machines, well-known complexity classes such as $\mathcal{NC}$, $\mathcal{P}$, $\mathcal{NP}$, etc. are characterized via logic programs.

In Chapter III, parallel computational complexity of logical queries on deductive databases is investigated. We address ourselves to subclasses of queries which can be effectively parallelized. We define a generalization of linear programs, called *non-confluent programs*, any of which has a derivation tree such that all of subgoals on it are distinct. We show that logical queries defined by all non-confluent programs are exactly equal to the queries in LOGCFL, the class of languages logspace reducible to context free languages. We also show a concrete procedure of parallelizing logical query programs. It transforms any naive non-confluent program into one with depth complexity $O(\log n)$.

In Chapter IV, we propose a new computation model of parallel data transfers on two-level storage. It is composed of processors with internal memory and a disk as a file server, which are connected by a shared bus. Computation time is measured by the number of data-transfer operations via the bus. The power of the model essentially depends on whether *broadcasting* to the bus is allowed or not. We show tight lower and upper bounds of the number of required data-transfer operations for sorting, permutation, FFT, matrix transposition and matrix multiplication, on both of models with and without broadcasting. Our results are multiprocessor extension of the bounds on the two-level memory model presented by Aggarwal and Vitter [AV88]. The proposed model gives a mathematical but realistic treatment of a distributed system composed of diskless workstations and a file server connected via LAN (e.g., Ethernet).

In Chapter V, use of extended storage of vector supercomputers as extended main memory is discussed. We propose a method which makes it possible for

users to perform large-scale computation with no awareness of data transfers between main memory and extended storage. We show algorithms for extracting declarations of huge arrays and references to them from an ordinary program, partitioning the arrays and allocating them on extended storage, and inserting codes which control data transfers between extended storage and temporary area on main memory. The transformation is done without changing the order of arithmetic operations in the original program. Vectorizability of the operations is also preserved by utilizing indirect vector addressing. To confirm the availability of this method, we have developed a FORTRAN preprocessor which does the program transformation at source-code level automatically. The transformed code of an LU-factorization program achieves about 50% of the execution speed when the original one is executed normally on main memory.

This thesis is concludeded with Chapter VI, where several open problems and suggestions to future works are also presented.

CHAPTER II

Parallel Computational Complexity of Logic Programs

1. Introduction

Logic programming languages have lately attracted considerable attention as languages for "fifth generation computers". In particular, since the execution of a logic program can be regarded as parallel computation, several parallel logic programming languages such as Concurrent Prolog [Sha83] and GHC [Ued85] are proposed in order to describe parallel processing. Moreover, many parallel computer architectures for processing logic programs are being developed, and some of them have been actually implemented. It becomes important to give a mathematical foundation to such a parallelism in logic programs, and to relate it to the conventional theory of parallel computation.

Shapiro [Sha84] showed that there is a close relationship between logic programs and alternating Turing machines (ATMs). In his formulation, a logic programs is regarded as an ATM, which is given a goal as an input string. The derivation of the program corresponds to the computation tree of the ATM, and the ATM accepts the input if and only if the program can solve the goal. He introduced three complexity measures for logic programs, namely, depth complexity, goal-size complexity and length complexity, and showed that these three are closely related to alternating time, space and tree-size respectively by simulating alternating Turing

machines and logic programs each other.

His result implies many relations among logic programs and conventional theoretical parallel computation models such as PRAMs and combinational circuits. According to his definition, however, goal-size complexity should be $\Omega(n)$ for the input length n; this is because "input" and "work space" are not distinguished. Corresponding alternating space complexity is also at least linear. For this reason, practically important classes like $\mathcal{P}$ (PTIME) cannot be characterized via logic programs. Štěpánek and Štěpánková [ŠŠ86] gave simulations between off-line alternating Turing machines and logic programs with sublinear space (or goal-size) complexity, for a special class of logic programs, which they call "logic programs with input". However, their result is not a natural improvement of Shapiro's since their logic programs are in a strongly restricted form. Ruzzo [Ruz80, Ruz81] showed that indexing alternating Turing machines (indexing ATMs), which are a "random access input" variation of ATMs, have a close relationship with practically important parallel complexity classes like $\mathcal{NC}$, but relationships between logic programs and indexing ATMs have never been discussed.

In this chapter, we improve Shapiro's (and also Štěpánek-Štěpánková's) results on relations between logic programs and alternating Turing machines, and characterize well-known parallel complexity classes in terms of logic programs. The main difference of our formulation from Shapiro's is that goal-size of logic programs is defined with the use of pointers taken into account. By our definition, the size of a term which occurs as a subterm in the initial goal clause can be estimated as the bit-length of the pointer representing it. Hence sublinear goal-size complexity is introduced naturally, just like space complexity of off-line Turing machines. This also make the random accessibility of indexing ATMs essential.

Two main theorems are derived using our new definition of goal-size complexity. One is an extension of Shapiro's first result to the case of sublinear space; logic

programs of goal-size $G(n)$ and depth $D(n)$ are simulated by indexing alternating Turing machines with space $G(n)$ and in time $D(n)G(n)$. This is achieved by extending Shapiro's simulation to the case of indexing ATMs. The other is an improvement of Shapiro's second result; indexing ATMs using space $S(n)$ and tree-size $Z(n)$ are simulated by logic programs with goal-size $S(n)$ and depth $\log Z(n)$, which is achieved by utilizing Ruzzo's techniques for simulating ATMs [Ruz80].

These two results imply a close relationship between logic programs and tree-size bounded alternating Turing machines. It can be said that simultaneous goal-size/depth of logic programs is the same as space/tree-size of indexing ATMs, with goal-size and space up to a constant factor and likewise depth and log(tree-size). Some well-known complexity classes such as LOGCFL, $\mathcal{NC}$, $\mathcal{P}$ and $\mathcal{NP}$ (NPTIME) can be characterized via logic programs. In particular, it is shown that context-free languages can be recognized by logic programs with depth $O(\log n)$ and goal-size $O(\log n)$ simultaneously.

In the next section, we will give several basic definitions on combinational circuits and alternating Turing machines. In Section 3, we will describe logic programs and define the complexity measures for them. Simulations between logic programs and indexing Turing machines will be presented in Section 4. In Section 5, classes of languages recognized by logic programs will be considered.

2. Basic concepts

2.1. Combinational circuits. A combinational circuit is a logic circuit which is composed of given computation elements (logic gates) and has no feedback loop in it. The fan-in (in-degree) of each computation element is restricted in a certain constant, while the fan-out (out-degree) of it is not restricted. Formal definitions of it is given as follows.

A *combinational circuit* is a directed acyclic graph, where each node (gate) has at most constant indegree d and is labeled by a Boolean function of d variables, or has indegree 0 and is labeled by "x" (an *input*). Nodes with outdegree 0 are *outputs*. *Size* is the number of nodes (gate count), which is considered to be a cost measure. *Depth* of a circuit is the length of the longest path from some input to some output. Computation time on a circuit is considered to be linearly proportional to the depth. If some output depends on all of the input, the depth of a circuit is $\Omega(\log n)$, where n is the number of input.

Next we define languages recognized by circuits. A *circuit family* $C = (c_1, c_2, \dots)$ is a infinite sequence of circuits, where c_n has n inputs and one output. C *recognizes* a language $A(\subseteq \{0,1\}^*)$ when $x_1 \dots x_n \in A$ iff the output value of c_n on input $(x_1, x_2, \dots, x_n) \in \{0,1\}^n$ is 1.

The *size* and the *depth* of C are defined as $Z(n)$ and $T(n)$, respectively, if c_n has at most $Z(n)$ gates and depth $T(n)$.

We restrict our attention to "uniform" circuit families. A circuit family C of size $Z(n)$ and depth $T(n)$ is called U_{E^*}-uniform if there exists an ATM recognizing the extended connection language of C in time $O(T(n))$ and space $O(\log Z(n))$ [Ruz81]. A U_{E^*}-uniform circuit is also logspace uniform. For each n (given in unary representation), the circuit description of c_n in a U_{E^*}-uniform circuit family C can be generated by an $O(\log n)$-space bounded Turing machine.

2.2. Alternating Turing machines. We assume familiarity with deterministic and non-deterministic Turing Machines (DTMs and NTMs, respectively). We also use *alternating Turing machines* (ATMs) [CKS81] as our computation model.

ATMs are a generalization of nondeterministic Turing machines described informally as follows. The states of an ATM are partitioned into "existential" and "universal" states. As with an NTM, we can view a computation of an ATM as a tree of configuration. A *full computation tree* of an ATM M on a string w is a

(possibly infinite) tree, each of whose nodes is labeled with a configuration of M on w, such that each node has all of the successors of that configuration as its descendants. (A leaf node is labeled with a configuration whose next-state transition set is empty.) A *computation tree* of M is a subtree of the full computation tree such that each non-leaf node labeled by universal configurations has all of the successors of that configuration as its descendants, while each non-leaf node labeled by existential configurations has only one of the successors of that configuration as the descendant.

An *accepting computation tree* is a finite computation tree all of whose leaf nodes are in accepting configurations. M accepts w if and only if there exists an accepting computation tree whose root node is labeled with the initial configuration of M on w. An NTM can be regarded as an ATM which has no universal state. Formal definitions of ATMs are found in [CKS81]. Note that we employ no "negating" state in the definition.

ATMs (which have a writable input tape) and off-line ATMs (which have a read-only input tape and some work tapes) are defined similarly as those for DTMs or NTMs. We also use a "random access input" variant of ATMs called *indexing ATMs*, introduced by Ruzzo [Ruz80]. An indexing ATM has no input head; instead it has a special *index tape* and special *read* states. Whenever it enters a read state with an integer i written on the index tape, it reads the i-th symbol of the input and transit to either an accepting or rejecting state. Thus, in $O(\log n)$ steps, it can read the character at any position of the input. Only a constant factor of loss in space and time is required in conversion of an off-line ATM to this normal form when space is at least $\Omega(\log n)$.

An ATM uses *time* $T(n)$ (*tree-size* $Z(n)$) if for all accepted inputs of length n there is an accepting computation tree of height $\leq T(n)$ (respectively, size $\leq Z(n)$). An ATM uses *space* $S(n)$ if for all accepted inputs there is an accepting

computation tree, each of whose nodes is labeled by a configuration using space $\leq S(n)$, and uses *alternation depth* $A(n)$ if for all accepted inputs there is an accepting computation tree, whose alternations of existential and universal configurations $\leq A(n)$.

We denote the class of languages recognized by indexing ATMs within space $O(S(n))$ and time $O(T(n))$ simultaneously, by $A\text{-}SpTi(S(n), T(n))$. Likewise, $A\text{-}SpSz(S(n), Z(n))$ denotes languages recognized by ATMs running in space $O(S(n))$ and tree-size $O(Z(n))$ simultaneously, and similarly $A\text{-}SpAl(S(n), A(n))$ etc.

2.3. Hierarchy of complexity classes. We are mostly concerned with classes of problems solvable very rapidly by a parallel computer with feasible number of processors, i.e., problems which can be computed by a uniform circuit with depth $O((\log n)^k)$ and polynomial size. Such a class is commonly called $\mathcal{NC}$. $\mathcal{NC}^k$ is the set of all problems solvable by a unform circuit family with depth complexity $O((\log n)^k)$ and size complexity $n^{O(1)}$. $\mathcal{NC}$ is defined as $\mathcal{NC} \stackrel{\text{def}}{=} \bigcup_k \mathcal{NC}^k$. Ruzzo showed a close relationship between uniform combinational circuits and indexing alternating Turing machines [Ruz81].

PROPOSITION II.1 (RUZZO 1981). *$\mathcal{NC}^k$ consists of all problems solvable by indexing ATMs in time $O((\log n)^k)$ and space $O(\log n)$, where n is the length of the input.*

Not a few known problems in $\mathcal{NC}$ are also in LOGCFL, which is a subclass of $\mathcal{NC}^2$. LOGCFL consists of all languages which are logspace reducible to the class of context free languages. (Here A is *logspace reducible* to B iff there is some logspace computable function f such that for all x, $x \in A$ iff $f(x) \in B$.) Ruzzo characterized LOGCFL as the class of problems recognizable by tree-size bounded ATMs [Ruz80].

PROPOSITION II.2 (RUZZO 1980). LOGCFL *consists of all problems solvable by ATMs in space* $O(\log n)$ *and tree-size* $n^{O(1)}$.

$\mathcal{AC}^k$ is the class of all problems solvable by an ATM in space $O(\log n)$ and alternation depth $O((\log n)^k)$. It is known that $\mathcal{NC}^k \subseteq \mathcal{AC}^k \subseteq \mathcal{NC}^{k+1}$ for any $k = 1, 2, \ldots$. Note that LOGCFL $\subseteq \mathcal{AC}^1$, and hence $\subseteq \mathcal{NC}^2$.

Hierarchy of these classes are

$$\textit{Regular Sets} \subset \mathcal{NC}^1 \subseteq \mathcal{DL} \subseteq \mathcal{NL} \subseteq \text{LOGCFL} \subseteq \mathcal{AC}^1 \subseteq \mathcal{NC}^2 \subseteq \cdots \subseteq \mathcal{NC} \subseteq \mathcal{P}$$

Here $\mathcal{DL}$ (DLOGSPACE) is the class solvable by a DTM in space $O(\log n)$, $\mathcal{NL}$ (NLOGSPACE) is the class solvable by an NTM in space $O(\log n)$, and $\mathcal{P}$ (PTIME) is the class solvable by a DTM in time $n^{O(1)}$ [Coo85].

3. Logic programs and their complexity

3.1. Logic programs. Let $\mathbf{F}$ be a finite set of *function symbols*, $\mathbf{P}$ be a finite set of *predicate symbols* and $\mathbf{V}$ be a set of *variables*. Each function symbol is characterized by its name and its arity. Zero-arity function symbols are called *constants*. Variables are distinguished by an initial capital letter.

Terms on $\mathbf{F} \cup \mathbf{V}$ are defined recursively as follows:

(1) A variable $X \in \mathbf{V}$ or a constant $a \in \mathbf{F}$ is a term.

(2) If $t_1, ..., t_k$ are terms and $f \in \mathbf{F}$ is a k-arity function symbol, $f(t_1, ..., t_k)$ is a term.

(3) All terms are generated by applying the above rules (1) and (2).

Let $p \in \mathbf{P}$ is a k-arity predicate symbol, and let $t_1, \ldots, t_k$ be terms. $p(t_1, \ldots, t_k)$ is called an *atomic formula*.

Let $\mathbf{T}$ be the set of terms on $\mathbf{F} \cup \mathbf{V}$. A *substitution* $\theta : \mathbf{V} \to \mathbf{T}$ is represented by a finite set of ordered pairs of terms and variables

$$\{ \langle X_1, t_1 \rangle, \langle X_2, t_2 \rangle, \ldots, \langle X_n, t_n \rangle \}.$$

X_i's are all distinct variables, and *theta* substitutes a variable X_i into a term t_i. Applying a substitution θ to a term t (namely applying it to all variables in t), we represent the resulting term by $t\theta$, and call it an *instance* of t. A substitution θ is called a *unifier* for two terms t_1 and t_2, if $t_1\theta = t_2\theta$. A unifier θ is said to be the *most general unifier* of t_1 and t_2 if $t_1\theta_1$ is an instance of $t_1\theta$ and $t_2\theta_1$ is an instance of $t_2\theta$ for any unifier θ_1 of t_1 and t_2. If two terms are unifiable then they have a unique most general unifier. Substitutions for atomic formulae are defined similarly.

Let A, B_1, ..., B_e ($e \geq 0$) be atomic formulae. A formula

$$A \leftarrow B_1, ..., B_e$$

is called a *Horn clause*, whose right side represents the conjunction of B_i's. A is called the *head* of the rule and $B_1, \ldots, B_e$ are called *subgoals* of it. We denote

$$A \leftarrow$$

when $e = 0$. A *logic program* is a finite set of Horn clauses. A logic program is shown in Fig. II.1 as an example. (It represents Peano's axioms of natural number system.)

We define a computation of a logic program. A conjunction of atomic formulae "$A_1, ..., A_m$", $m \geq 0$, is called a *goal clause*, or simply a *goal*. When $m = 1$, "A_1" is called a *unit goal*. When $m = 0$, we denote it $\square$ and call it an *empty goal*.

Let $\boldsymbol{N}$ = "$A_1, A_2, ..., A_m$", $m \geq 0$, be a (conjunctive) goal and $\boldsymbol{C}$ = "$A \leftarrow B_1, ..., B_e$", $e \geq 0$, be a Horn clause such that A and A_i for some i ($1 \leq i \leq m$) are unifiable via a substitution θ. Then

$$\boldsymbol{N}' = (A_1, ..., A_{i-1}, B_1, ..., B_e, A_{i+1}, ..., A_m)\theta$$

is said to be *derived* from $\boldsymbol{N}$ and $\boldsymbol{C}$ with θ.

$$\mathbf{F} = \{s, 0\}$$

$$\mathbf{P} = \{p_{\text{eq}}, p_{sum}\}$$

$$\mathbf{V} = \{X, Y, Z, W\}$$

$$\begin{aligned} P = \{ \quad p_{\text{eq}}(0,0) &\leftarrow && \text{(i)} \\ p_{\text{eq}}(s(X), s(Y)) &\leftarrow p_{\text{eq}}(X,Y) && \text{(ii)} \\ p_{\text{sum}}(X, 0, Y) &\leftarrow p_{\text{eq}}(X,Y) && \text{(iii)} \\ p_{\text{sum}}(X, s(Y), Z) &\leftarrow p_{\text{sum}}(X,Y,W),\ p_{\text{eq}}(Z, s(W)) && \text{(iv)} \end{aligned}$$

$$\}$$

FIGURE II.1. A logic program (example).

Let P be a logic program and $\boldsymbol{N}$ be a goal. A *derivation* of $\boldsymbol{N}$ from P is a (possibly infinite) sequence of triples $\langle \boldsymbol{N}_i, \boldsymbol{C}_i, \theta_i \rangle$, $i = 0, 1, \ldots$, such that $\boldsymbol{N}_i$ is a goal, $\boldsymbol{C}_i$ is a clause in P, θ_i is a substitution, $\boldsymbol{N}_0 = \boldsymbol{N}$ and $\boldsymbol{N}_{i+1}$ is derived from $\boldsymbol{N}_i$ and $\boldsymbol{C}_i$ with substitution θ_i, for all $i \geq 0$.

A derivation of $\boldsymbol{N}$ from P is called a *refutation* of $\boldsymbol{N}$ from P if $\boldsymbol{N}_l = \square$ (the empty goal) for some $l \geq 0$. Such a derivation is finite and of length l. If there is a refutation of a goal A from a program P, we say that P *solves* A. Let $\boldsymbol{R}$ be a refutation of A from P. The *refutation tree* of $\boldsymbol{R}$ is a tree of unit goals, which is defined as follows:

(1) The root of the tree is A.

(2) Leaves are empty goals.

(3) At each step of derivation $\langle \boldsymbol{N}_i, \boldsymbol{C}_i, \theta_i \rangle$, if $\boldsymbol{C}_i =$ "$A_i \leftarrow B_1, \ldots, B_e$", and unit goal A_{ij} in $\boldsymbol{N}_i$ is unified with A_i by θ_i, then the node A_i has directed edges to all $B_i\theta_i$'s.

A refutation and its refutation tree of the program P in Fig. II.1 is shown in Fig. II.2.

The *Herbrand universe* of P, $H(P)$, is the set of all variable-free terms composed of function symbols that occurs in P. The *Herbrand base* of P, $HB(P)$, is the set of all atomic formulae composed of predicate symbols that occurs in P and terms in $H(P)$. We define the *interpretation* of P, $I(P)$, to be the set

$$\{A \in HB(P) | P \text{ solves } A.\}.$$

3.2. Complexity measures for logic programs. First we describe complexity measures of logic programs introduced by Shapiro [Sha84].

Let P be a logic program and A_0 be a unit goal. We regard P as a machine which determines whether P solves a given goal or not. Here a goal A_0 is a input string for P, and P accepts A_0 if and only if P solves A_0. A goal is to a program what a string on the input tape is to a Turing machine. The interpretation $I(P)$ is considered to be the "language" recognized by the program P. (P itself corresponds to, as it were, a finite control of the Turing machine.)

Assume that P solves A_0 with a refutation $\boldsymbol{R}$. We regard $\boldsymbol{R}$ as a computation of P for input A_0. The *length* of $\boldsymbol{R}$ is defined as the number of nodes in the refutation tree of $\boldsymbol{R}$, and the *depth* of $\boldsymbol{R}$ is defined as the height of the refutation tree. The *goal-size* of $\boldsymbol{R}$ is defined as the maximum size of any node in the refutation tree, where the size of goal is the number of symbols in its prefix notation. We say that a logic program P is of *goal-size complexity* $G(n)$, if any goal A_0 in $I(P)$ of size n has a refutation from P of goal-size $\leq G(n)$. *Depth complexity* and *length complexity* of P are defined similarly.

Intuitively, the three complexity measures, depth, length and goal-size, are considered to be *"parallel" computation time*, *"serial" computation time* and *space*, respectively, required in computing $\boldsymbol{R}$. In fact, Shapiro showed the following cor-

Goal: "$p_{\text{sum}}(s(0), s(0), s(s(0)))$"

$\langle$ "$p_{\text{sum}}(s(0), s(0), s(s(0)))$", "$p_{\text{sum}}(X_1, s(Y_1), Z_1) \leftarrow p_{\text{sum}}(X_1, Y_1, W_1),\ p_{\text{eq}}(Z_1, s(W_1))$",
$\{\langle X_1, s(0)\rangle, \langle Y_1, 0\rangle, \langle Z_1, s(s(0))\rangle\}\rangle$

$\langle$ "$p_{\text{sum}}(s(0), 0, W_1),\ p_{\text{eq}}(s(s(0)), s(W_1))$", "$p_{\text{sum}}(X_2, 0, Y_2) \leftarrow p_{\text{eq}}(X_2, Y_2)$",
$\{\langle X_2, s(0)\rangle, \langle Y_2, W_1\rangle, \}\rangle$

$\langle$ "$p_{\text{eq}}(s(0), W_1),\ p_{\text{eq}}(s(s(0)), s(W_1))$", "$p_{\text{eq}}(s(X_3), s(Y_3)) \leftarrow p_{\text{eq}}(X_3, Y_3)$",
$\{\langle X_3, 0\rangle, \langle W_1, s(Y_3)\rangle, \}\rangle$

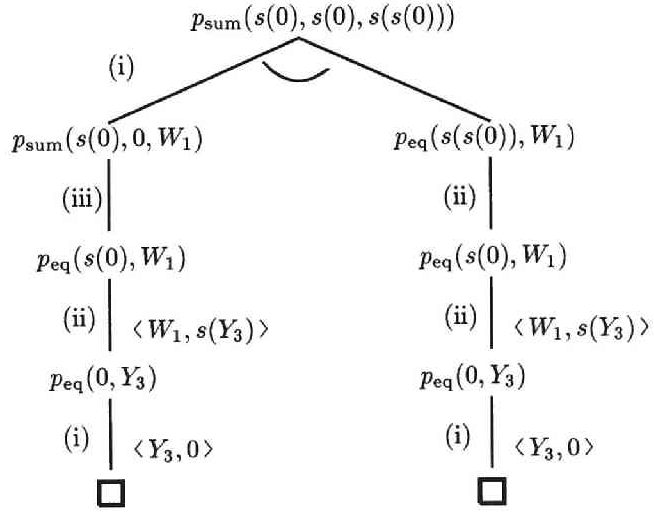
$\langle$ "$p_{\text{eq}}(0, Y_3),\ p_{\text{eq}}(s(s(0)), s(s(Y_3)))$", "$p_{\text{eq}}(0, 0) \leftarrow$ ", $\{\langle Y_3, 0\rangle\}\rangle$

$\langle$ "$p_{\text{eq}}(s(s(0)), s(s(0)))$", "$p_{\text{eq}}(s(X_4), s(Y_4)) \leftarrow p_{\text{eq}}(X_4, Y_4)$",
$\{\langle X_4, s(0)\rangle, \langle Y_4, s(0)\rangle\}\rangle$

$\langle$ "$p_{\text{eq}}(s(0), s(0))$", "$p_{\text{eq}}(s(X_5), s(Y_5)) \leftarrow p_{\text{eq}}(X_5, Y_5)$", $\{\langle X_5, 0\rangle, \langle Y_5, 0\rangle\}\rangle$

$\langle$ "$p_{\text{eq}}(0, 0)$", "$p_{\text{eq}}(0, 0) \leftarrow$ ", $\varnothing\rangle$

(a) A refutation of $\boldsymbol{P}$ in Fig. II.1.



(b) The associated refutation tree of (a).

FIGURE II.2. A refutation and its refutation tree.

respondence between logic programs and alternating Turing machines [Sha84].

PROPOSITION II.3 (SHAPIRO 1984). *Let P be a logic program of depth complexity $D(n)$, goal-size complexity $G(n)$ and length complexity $L(n)$. Then there exists an ATM M recognizing $I(P)$ such that M operates in time $O(D(n)G(n))$, space $O(G(n))$ and tree-size $O(L(n)G(n))$.*

NOTE. Any logic program has goal-size complexity of $G(n) = \Omega(n)$.

PROPOSITION II.4 (SHAPIRO 1984). *Let M be an ATM with a writable input tape that recognizes a language L in time $T(n)$, space $S(n)$ and tree-size $Z(n)$. Then there exists a logic program P of depth complexity $O(T(n))$, goal-size complexity $O(S(n))$ and length complexity $O(Z(n))$ such that*

$$L = \{w \in \Sigma^* | p_{accept}(w) \text{ is in } I(P)\}.$$

NOTE. Note that any ATM with a writable input tape has space complexity of $S(n) = \Omega(n)$ and time complexity of $T(n) = \Omega(n)$.

It is quite interesting that goal-size is linearly related to alternating space, and depth and length are related to alternating time and alternating tree-size respectively, up to the goal-size factor. However, in Shapiro's definition of goal-size, "input" and "work space" required in a computation are not distinguished, and therefore goal-size is at least n for the input length n. Thus the minimum class of languages defined in terms of logic-programs is APTIME = PSPACE! To cope with this deficiency, we introduce a new definition of goal-size.

Let $\boldsymbol{D}$ be a derivation of a unit goal A_0 from a program P. Consider a unit goal $A\theta^*$, where A be a unit goal close that occurs in $\boldsymbol{D}$ and θ^* is the successive application of all substitutions in $\boldsymbol{D}$. If a term t in $A\theta^*$ also occurs in A_0, t is called a *read-only* term. A read-only term which is not a subterm of any other read-only term is called a *primary read-only* term.

Let n be the number of function symbols in the initial goal A_0. The size of a primary read-only term is defined as the smaller of $\lceil \log_2 n \rceil$ and the number of function symbols occur in it in its textual representation. The size of a goal in $\boldsymbol{R}$ is defined as the sum of the size of all primary read-only terms and the number of function symbols which are not in read-only terms. The goal-size of a refutation is defined as the maximum size of all nodes in the refutation tree of $\boldsymbol{R}$. According to this new definition, the goal-size complexity of a program is defined similarly to the original one.

The new definition of goal-size is based on the fact that substring of an input string A_0 can be represented as a pointer to A_0 (the address of it in binary representation). Obviously our definition of the goal-size complexity is a natural extension of Shapiro's definition.

4. Alternating Turing machines and logic programs

4.1. Simulating logic programs with ATMs. We describe an algorithm for simulating a logic program by an indexing alternating Turing machine, and extend the result of Proposition II.3 to logic programs with sublinear goal-size complexity.

THOEREM II.5. *A logic program of goal-size complexity $G(n) = \Omega(\log n)$, depth complexity $D(n) = \Omega(\log n)$, length complexity $L(n)$, can be simulated by an indexing ATM in time $O(D(n)G(n))$, space $O(G(n))$, tree-size $O(L(n)(G(n) + n \log n))$ and alternation depth $O(D(n))$.*

Goal-size is linearly related to the alternating space, depth is related to the alternating time up to the goal-size factor (indeed it is linearly related to the alternation depth) and length is related to the alternating tree-size up to the polynomial factor.

The proof of this is given as an algorithm by which a indexing alternating Turing machine determines whether a logic program P solves a unit goal A_0 or not. It is

a modification of what Shapiro showed in [Sha84].

Without loss of generality, we may assume that arity of any function symbol is at most 2. Consider Algorithm SIMULATE1 shown in Fig. II.3. DERIVE is a procedure for the derivation and UNIFY is a subroutine for unification. In our simulation, goals are stored on the work tape of the indexing ATM, where ordinary terms are represented in prefix notation and primary read-only terms may be represented as a pointer to the input string.

UNIFY is a function which returns the most general unifier of a variable-free term t and a term t'. First we consider the case that t is an ordinary term. If t' is a variable X, it returns the substitution $\{ \langle X, t \rangle \}$ (in line (U1)). If t' is a constant a and $t = a$, it returns $\varnothing$ (in line (U2)). If t' is $f(t'_1)$ and t is $f(t_1)$, where f is a 1-arity function symbol and t'_1 and t_1 are terms, it returns the result of UNIFY(t_1, t'_1) recursively (in line (U3)). If t' is $g(t'_1, t'_2)$ and t is $g(t_1, t_2)$, where g is a 2-arity functor and t'_1, t'_2, t_1, t_2 are terms, it scans the term $g(t_1, t_2)$ represented in prefix notation on the work tape, guesses the address of the second argument t_2 nondeterministically (in line (U4)), and verifies it using universal branches (in line (U5)). Next it calls the subroutines UNIFY(t_1, t'_1) and UNIFY(t_2, t'_2) (in line (U6)), verifies if no variable is substituted in two or more inconsistent way (in line (U7)), and returns the union of them (in line (U8)). In every other case, viz. when these two terms are not unifiable, UNIFY "reject" the computation and all of the associated universal branches also fail.

It is almost similar in the case that t' is a primary read-only term represented as a pointer, except that arguments are passed by their addresses. In case that $t' = g(t'_1, t'_2)$, it expects that $t = g(t_1, t_2)$ and guesses the address of t_2 on the input tape nondeterministically (in line (U4)), instead of scanning the work tape.

Procedure DERIVE chooses a Horn clause $\boldsymbol{C}$ = "$A' \leftarrow B_1, ..., B_e$" ($e \geq 0$) in P nondeterministically (in line (R1)) and get the most general unifier σ by

Algorithm SIMULATE1

Given: a logic proaram P;

Input: a unit goal A;

Output: whether P proves A or not;

```
function UNIFY( t: a variable free term, t': a term ) : a substitution;
    begin
        case( t' ) of
(U1)        a variable "X": UNIFY := { < X, t > };
(U2)        a constant "a": if t= "a" then UNIFY := ∅ else reject;
            an 1-arity function symbol "f(t'_1)":
(U3)            if ( t= "f(t_1)" ) then UNIFY := UNIFY( t_1, t'_1 ) else reject;
            a 2-arity function symbol "g(t'_1, t'_2)":
                if ( t= "g(t_1, t_2)" ) then
                    begin
(U4)                    Guess the address of t_2 (existential branch);
(U5)                    confirm the guess (U4) is correct or not (universal branch);
(U6)                    UNIFY1 := UNIFY( t_1, t'_1 );  UNIFY2 := UNIFY( t_2, t'_2 );
(U7)                    if ( UNIFY1 and UNIFY2 are inconsistent ) then reject;
(U8)                    UNIFY := UNIFY1 ∪ UNIFY2;
                    end
                else reject;
        end { of case }
    end { of function UNIFY };

procedure DERIVE ( A: a unit goal );
    begin
(R1)    Choose a clause C = "A' ← B_1, ..., B_k" (k ≥ 0) in P (existential branch);
(R2)    σ := UNIFY( A, A' );
(R3)    Guess a substitution θ (existential branch) such that
         all remaining variables in Cσ are substituted to variable free terms;
(R4)    for all i ∈ {1, ..., k} parallel do (universal branch)
            DERIVE ( B_iσθ );
    end { of procedure DERIVE } ;

begin { of main routine }
    Output "YES" if DERIVE( A ) is successfully end;
end;
```

FIGURE II.3. An algorithm for simulating a logic program by an indexing ATM.

calling Function UNIFY (in line (R2)). Suppose A and A' are unifiable. Since no variable occurs in goal A, all variables occur in A' are substituted by variable-free terms via σ. Next it guesses a variable-free substitution θ of variables which are not substituted via σ (in line (R3)). All variables occur in $\boldsymbol{C}$ are substituted by variable-free terms via $\sigma\theta$. Finally, it asks if $B_i\sigma\theta$ can be derived by P for each B_i using universal branches (in line (R4)).

Let us consider how much space, time and tree-size are required in this simulation, by examining how the refutation $\boldsymbol{R}$ of A_0 from P is simulated by an indexing ATM. Let n be the length of A_0, and let $G(n)$, $D(n)$ and $L(n)$ be the goal-size, depth and length of $\boldsymbol{R}$ respectively. Space, time and tree-size required in each call of Function UNIFY are all $O(\log n + G(n)) = O(G(n))$, and required alternation depth is at most 2, except for the verifications of consistency (in lines (U5) and (U7)). Since the clause A' is independent of n, the depth of recursive calls of Function UNIFY is at most a constant independent of n, and therefore each call of Procedure DERIVE requires $O(G(n))$-space, $O(G(n))$-time, $O(G(n))$-tree-size and at most constant alternation depth. Thus the simulation of $\boldsymbol{R}$ is carried out in space $O(G(n))$, time $O(D(n)G(n))$, tree-size $O(L(n)G(n))$ and alternation depth $O(D(n))$ except for the verification steps (in lines (U5) and (U7)). Verification of consistency is essentially equivarent to recognition of a context-free language, and can be carried out in space $O(\log n)$, time $O((\log n)^2)$, tree-size $O(n(\log n))$ and alternation depth $O(\log n)$ [Ruz80]. Thus the total space, time, tree-size and alternation depth required in the simulation are $O(G(n) + \log n)$, $O(D(n)G(n)+(\log n)^2)$, $O(L(n)(G(n)+n\log n))$ and $O(D(n)+\log n)$ respectively. We now get the threorem.

4.2. Simulating ATMs with logic programs. We describe an algorithm for simulating indexing alternating Turing machines by logic programs. This algorithm is based on a simulation technique introduced by Ruzzo (1980) for simulating

an $S(n)$-space and $Z(n)$-tree-size bounded ATM (which may be indexing, off-line or ordinary) by an $O(S(n)\log Z(n))$-time bounded and $O(S(n))$-space bounded indexing ATM.

A computation *fragment* of an $S(n)$-space and $Z(n)$-tree-size bounded ATM M is an ordered pair (r, L), where r is a configuration of M using space $\leq S(n)$, and L is a set of such configurations. A fragment is *realizable* if there is a computation tree of M with root r whose leaves are either accepting or in L. An algorithm for deciding realizability of a fragment (r, L). Algorithm SIMULATE2, is shown in Fig. II.4.

PROPOSITION II.6 (RUZZO 1980). *If ATM M accepts w within tree-size $\leq Z(n)$, $REAL(r_0, \varnothing)$ returns "true" with maximum depth of recursions $O(\log Z(n))$.*

Ruzzo also showed that it is sufficient to consider fragments whose corresponding computation trees have not more than three non-accepting leaves in Algorithm SIMULATE2.

Next we describe a logic program which simulates Algorithm SIMULATE2. We may assume that M is an indexing ATM which has one work tape and that the index tape alphabet is $\{0, 1\}$. In our simulation, an input string $w = x_1 x_2 ... x_n \in \Sigma^*$ of length n is represented by a term

$$\cdot(\ldots(\cdot(\cdot(x_1, x_2), \cdot(x_3, x_4)), \cdots \cdot(\cdot(x_n, \$), \cdot(\$, \$))\ldots)$$

in complete-binary-tree form of height $\lceil \log_2 n \rceil$ as shown in Fig. 4 (a), where "$\cdot$" is a 2-arity function symbol and "\$" is a constant. A configuration of indexing ATM M is represented by a term

$$q(w_{\text{cur}}, t_{left}, t_{right})$$

where w_{cur} is a subtree of w obtained by traversing a complete binary tree w according to w_{index}, and t_{left} and t_{right} represent the left side (including the head

Algorithm SIMULATE2

Given: an indexing alternating Turing machine M.

Input: a string $w \in \Sigma^*$;

Output: whether M accepts w or not;

function REAL(r: a configuration of M,
L: a set of configurations of M): Boolean;

begin

(F1) **if** the fragment (r, L) is realizable within tree-size ≤ 3 **then**
REAL := true
else
begin

(F2) Guess a configuration of M, s,
and sets of configurations of M, L' and L''
s.t. $L' \subseteq L$, $L'' \subseteq L$ and $L = L' \cup L''$
(existential branch) ;

(F3) REAL := (REAL(r, $L' \cup \{s\}$) $\wedge$ REAL(s, L''))
(universal branch);
end

end {*of REAL*};

begin
Output "YES" if REAL(r_0, $\varnothing$) = true,
where r_0 is an initial configuration of M with input w;
end.

FIGURE II.4. An algorithm for simulating an ATM.

position) and the right side of the content of the work tape respectively. (See an example shown in Fig. II.5(b).)

Let us consider the logic program P_{real} shown in Fig. 5. Obviously P_{real} behaves quite similar to Algorithm SIMULATE2. From PropositionII.6, if M accepts w within space $\leq S(n)$ and tree-size $\leq Z(n)$, then there exists a refutation of

$$p_{\text{accept}}(q_0(w))$$

from P_{real} with depth $O(\log Z(n))$, goal-size $O(Z(n))$ and length $O(Z(n))$. Thus the second theorem follows:

THOEREM II.7. *Let M be an ATM which accepts a language L in space $S(n) \geq \Omega(\log n)$ and tree-size $Z(n)$. Then there exists a logic program P of depth complexity $O(\log Z(n))$, goal-size complexity $O(S(n))$ and length complexity $O(Z(n))$ such that*

$$\{w \in \Sigma^* |\ p_{accept}(w) \text{ is in } I(P).\}$$

is a language recognized by M.

Alternating space is linearly related to the goal-size. Alternating tree-size is linearly related to the length and logarithm of alternating tree-size is linearly related to the depth. The next corollary follows immediately from the theorem.

COROLLARY II.8. *Let M be an ATM that accepts a language L in time $T(n)$, space $S(n) \geq \Omega(\log n)$ and tree-size $Z(n)$. Then there exists a logic program P of depth complexity $O(T(n))$, goal-size complexity $O(S(n))$ and length complexity $O(Z(n))$ such that $\{w \in \Sigma^* | accept(w)$ is in $I(P).\}$ is a language recognized by M.*

Remark. Note that $T(n) \geq \Omega(\log Z(n))$. □

Since Corollary II.8 subsumes Proposition II.4, Theorem II.7 is an improvement of Proposition II.4.

Program P_{real}

$$
\begin{aligned}
p_{\text{accept}}(W) \leftarrow & \; p_{\text{realizable}}(q_0(W,\$,\$), f_L(\$,\$,\$)) \\
p_{\text{realizable}}(Y, f_L(Y_1,Y_2,Y_3)) \leftarrow & \; p_{\text{realizable}}(Y, f_L(Z,Y_2,Y_3)), \\
& \; p_{\text{realizable}}(Z, f_L(Y_1,Y_2,Y_3)) \\
p_{\text{realizable}}(Y, f_L(Y_1,Y_2,Y_3)) \leftarrow & \; p_{\text{realizable}}(Y, f_L(Y_2,Y_1,Y_3)) \\
p_{\text{realizable}}(Y, f_L(Y_1,Y_2,Y_3)) \leftarrow & \; p_{\text{realizable}}(Y, f_L(Y_3,Y_2,Y_1)) \\
p_{\text{realizable}}(Y, f_L(Y_1,Y_2,Y_3)) \leftarrow & \; p_{\vee}(Y,Z), \\
& \; p_{\text{realizable}}(Z, f_L(Y_1,Y_2,Y_3)) \\
p_{\text{realizable}}(Y, f_L(Y_1,Y_2,Y_3)) \leftarrow & \; p_{\wedge}(Y,Y_1,Y_2), \\
& \; p_{\text{realizable}}(Y_1, f_L(\$,\$,\$)), \\
& \; p_{\text{realizable}}(Y_2, f_L(\$,\$,\$)) \\
p_{\text{realizable}}(Y, f_L(Y_1,Y_2,Y_3)) \leftarrow & \; p_{\vee}(Y,Y_1), \\
& \; p_{\text{realizable}}(Y_1, f_L(\$,\$,\$)) \\
p_{\wedge}(q_{\wedge i}(X), q_a(X), q_b(X)) \leftarrow & \\
p_{\vee}(q_{\vee i}(X), q_c(X)) \leftarrow & \\
p_{\vee}(q_1(\cdot\,(W_1,W_2),T_1,T_2), q_2(W_1,T_1,T_2)) \leftarrow & \\
p_{\vee}(q_1(\cdot\,(W_1,W_2),T_1,T_2), q_3(W_2,T_1,T_2)) \leftarrow & \\
p_{\vee}(q_1(W, f_{c_1}(T_1), f_{c_2}(T_2)), q_4(W, f_{c_2}(f_{c'}(T_1)), T_2)) \leftarrow & \\
p_{\vee}(q_1(W, f_{c_1}(f_{c_3}(T_1)), T_2), q_5(W, f_{c_3}(T_1)), f_{c''}(T_2)) \leftarrow & \\
p_{\text{realizable}}(q_6(0, T_1.T_2)) \leftarrow & \\
p_{\text{realizable}}(q_7(1, T_1.T_2)) \leftarrow &
\end{aligned}
$$

FIGURE II.5. A program for simulating an ATM.

5. Languages recognized by logic programs

We examine the relations among classes of languages recognized by logic programs and many well-known complexity classes.

We denote the class of languages recognized by logic programs within goal-size $O(G(n))$, depth $O(D(n))$ and length $O(L(n))$ simultaneously, by $L\text{-}GzDpLn(G(n), D(n), L(n))$. Likewise, $L\text{-}GzDp(G(n), D(n))$ denotes languages recognized by logic programs within goal-size $O(G(n))$ and depth $O(D(n))$ simultaneously, and similarly for $L\text{-}GzLn(G(n), L(n))$ etc.

THOEREM II.9. *For $D(n) \geq \Omega(\log n)$ and $G(n) \leq 2^{O}(D(n))$,*

$$\begin{aligned} L\text{-}GzDp(G(n), D(n)) &= L\text{-}GzLn(G(n), 2^{O(D(n))}) \\ &= A\text{-}SpSz(G(n), 2^{O(D(n))}). \end{aligned}$$

PROOF. From Proposition II.3 and Theorem II.5,

$$\begin{aligned} L\text{-}GzLn(G(n), 2^{O(D(n))}) &\subseteq A\text{-}SpSz(G(n),\ 2^{O(D(n))} \cdot (G(n) + n\log^2 n)) \\ &= A\text{-}SpSz(G(n), 2^{O(D(n))}). \end{aligned}$$

From Theorem II.7,

$$A\text{-}SpSz(G(n), 2^{O(D(n))}) \subseteq L\text{-}GzDpLn(G(n), D(n), 2^{O(D(n))}).$$

It is obvious that, if the depth of the refutation is $D(n)$, the length of a refutation is at most $2^{O(D(n))}$, i.e.

$$L\text{-}GzDp(G(n), D(n)) \subseteq L\text{-}GzDpLn(G(n), D(n), 2^{O(D(n)}).$$

□

COROLLARY II.10. *The class of languages recognized by logic programs within goal-size $O(\log n)$ and depth $O(\log n)$ is* LOGCFL, *i.e. the class of languages*

log-space reducible to context free languages.

$$\begin{aligned} \textit{L-GzDp}(\log n, \log n) &= \textit{L-GzLn}(\log n, 2^{O(\log n)}) \\ &= \textit{A-SpSz}(\log n, 2^{O(\log n)}) \\ &= \textit{LOGCFL}. \end{aligned}$$

Remark. The corollary immediately follows from Theorem II.9 for $G(n) = O(\log n)$ and $D(n) = O(\log n)$. □

Similarly, the following result follows immediately.

$$\begin{aligned} \textit{L-GzDp}(\log n, (\log n)^{O(1)}) &= \textit{L-GzLn}(\log n, 2^{(\log n)^{O(1)}}) \\ &= \textit{A-SpSz}(\log n, 2^{(\log n)^{O(1)}}) \\ &= \mathcal{NC}, \end{aligned}$$

$$\begin{aligned} \textit{L-GzDp}(\log n, n^{O(1)}) &= \textit{L-GzLn}(\log n, 2^{n^{O(1)}}) \\ &= \textit{A-SpSz}(\log n, 2^{n^{O(1)}}) \\ &= \mathcal{P}, \end{aligned}$$

$$\begin{aligned} \textit{L-GzDp}(n^{O(1)}, \log n) &= \textit{L-GzLn}(n^{O(1)}, n^{O(1)}) \\ &= \textit{A-SpSz}(n^{O(1)}, n^{O(1)}) \\ &= \mathcal{NP}. \end{aligned}$$

Hierarchy of complexity classes are shown in Fig. II.6.

6. Considerations

We have modified the definition of complexity of logic programs introduced by Shapiro, and have shown a relationship between the complexity of logic programs and that of alternating Turing machines. In particular, logic programs are closely related to tree-size bounded alternating Turing machines. We can say that logic

$$L\text{-}GzDp(n^{O(1)}, n^{O(1)}) = A\text{-}SpSz(n^{O(1)}, 2^{n^{O(1)}}) = \text{PSPACE}$$
$$= L\text{-}GzLn(n^{O(1)}, 2^{n^{O(1)}}) = A\text{-}SpTi(n^{O(1)}, n^{O(1)})$$

$$\|$$

$$L\text{-}GzDp(n^{O(1)}, \log n) = A\text{-}SpSz(n^{O(1)}, n^{O(1)}) = \mathcal{NP}$$
$$= L\text{-}GzLn(n^{O(1)}, n^{O(1)})$$

$$\|$$

$$L\text{-}GzDp(\log n, n^{O(1)}) = A\text{-}SpSz(\log n, 2^{n^{O(1)}}) = \mathcal{P}$$
$$= L\text{-}GzLn(\log n, 2^{n^{O(1)}})$$

$$\|$$

$$= L\text{-}GzDp(\log n, (\log n)^{O(1)}) = A\text{-}SpSz(\log n, 2^{(\log n)^{O(1)}}) = \mathcal{NC}$$
$$L\text{-}GzLn(\log n, 2^{(\log n)^{O(1)}}) = A\text{-}SpTi(\log n, (\log n)^{O(1)})$$

$$\|$$

$$A\text{-}SpTi(\log n, (\log n)^2) = \mathcal{NC}^2$$

$$\|$$

$$A\text{-}SpAl(\log n, \log n) = \mathcal{AC}^1$$

$$\|$$

$$L\text{-}GzDp(\log n, \log n) = A\text{-}SpSz(\log n, 2^{O(\log n)}) = \text{LOGCFL}$$
$$= L\text{-}GzLn(\log n, 2^{O(\log n)})$$

$$\|$$

$$\mathcal{NL}$$

$$\|$$

$$\mathcal{DL}$$

$$\|$$

$$A\text{-}SpTi(\log n, \log n) = \mathcal{NC}^1$$

$$\|$$

Regular Sets

FIGURE II.6. Hierarchy of complexity classes with respect to logic program complexity.

programs are a surprisingly good model of parallel computation. Indeed, logics program can be considered as alternating Turing machines whose existential branches and universal branches are not symmetric.

$\mathcal{NC}$ is a very attractive class, because all problems in it are solvable very rapidly by circuits with polynomial gates. We have given a characterization of $\mathcal{NC}$ in terms of logic programs. Another characterization of $\mathcal{NC}$ via logic programs is shown in the next chapter.

Theorem II.9 means that any logic programs of length complexity $L(n)$ (whose depth complexity is $O(L(n))$) can be simulated by a logic program with depth complexity $O(\log L(n))$ and length complexity $L(n)^{O(1)}$. This may be regard as conversion from "serial" logic programs into parallel form. In the next chapter, we show a concrete procedure for converting a program into a parallelized equivalent.

CHAPTER III

Parallelizing Logical Query Programs

1. Introduction

Logic programs without function symbols, called *logical query programs* or *Datalog* programs, have been studied as a query language of deductive databases. In this chapter, we argue theoretically on the computational complexity of logical query programs.

First-order queries are queries expressible in first-order relational calculus with finite domains [Cod70]. First-order queries are in $\mathcal{AC}^0$, and thus there exist so many queries that are not expressible in first-order logic [AU79]. Fixpoint queries are obtained by augmenting first-order relational calculus with the least fixed point operator μ [CH84]. The class of fixpoint queries is exactly equal $\mathcal{P}$ (PTIME), if total ordering $\leq$ is defined on the domains [Imm86]. Another additional operator for first-order queries is *transitive closure*. Immerman showed that the class of first-order queries with total ordering plus transitive closure is equal to $\mathcal{NL}$ (NLOGSPACE) [Imm87, Imm88].

Logical queries are queries expressible in logical query programs. Formalization of logic query programs as a theoretical model of computation was given by Ullman and Van Gelder [UV88]. A logical query is defined by function-free clauses (called IDB (Intensional Database) rules) and a a set of predicates instantiated with

constants (called EDB (Extensional Database) facts) given as an input instance. The size of *fringes* of a derivation is used as a complexity measure. They showed a PRAM algorithm which computes the query defined by a logical query program, and showed that queries with the polynomial fringe property is in $\mathcal{NC}$. They also show that some logical query programs are logspace complete for $\mathcal{P}$ ($\mathcal{P}$-complete).

However, there exist some queries which are in $\mathcal{P}$ but cannot be defined by logical query programs, because of the absence of both negation and total ordering.

We here investigate logical query programs with total ordering plus negation of EDB facts. We show that the class of queries defined by such logical query programs is exactly equal to that of queries in $\mathcal{P}$, by characterizing logical query programs as logspace bounded alternating Turing machines. The fringe complexity of a program is related to the tree-size complexity of an ATM. Some results obtained on conventional computation models can be translated in terms of logic programs. We also show a concrete procedure to parallelize a program with polynomial fringe property into equivalent one with logarithmic depth complexity.

Next we address ourselves to subclasses of queries which can effectively parallelized. *A non-confluent program* is a logical query program, which has a derivation tree such that all of subgoals on it are distinct. We show that logical queries defined by non-confluent programs are exactly equal to the queries in LOGCFL, the class of languages logspace reducible to context free languages (CFL). The polynomial fringe property is related to non-confluency.

In Section 2, we will describe some formal definitions on logical query programs as our computation model. In Section 3, it will be estimated how much power is attained in use of negation of EDB facts and total ordering. Simulations between logical query programs and logspace bounded alternating Turing machines will be presented in Section 4. In Section 5, we will show a procdure of "parallelizing" logical query programs. Linear programs, non-confluent programs and the

polynomial fringe property will be investigated in Section 6.

2. Definitions

2.1. EDB facts, EDB instances, and queries. Let $\mathbf{V}$ be a finite set of variables, and let $\mathbf{C} = \{c_1, c_2, \dots\}$ be a countably infinite sequence (totally ordered set) of constants. $\mathbf{C}$ is called the *universe*, in association with the Herbrand universe of a logic program. A *function-free term* is either a variable or a constant. We abbreviate a function-free term to just a term hereafter. A *domain* is a leading subsequence of the universe $\mathbf{C}$, $\mathbf{C}_n = \{c_1, \dots, c_n\}$, for some $n \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers (non-negative integers).[1]

Let $\mathbf{P}_{\text{IDB}}$ and $\mathbf{P}_{\text{EDB}}$ be mutually disjoint sets of predicate symbols. Predicate symbols in $\mathbf{P}_{\text{IDB}}$ are called *IDB (Intensional Database) predicate symbols*, and predicate symbols in $\mathbf{P}_{\text{EDB}}$ are called *EDB (Extensional Database) predicate symbols*. We use $\{p_0, p_1, \dots\}$ to denote IDB predicate symbols, $\{q_0, q_1, \dots\}$ to denote EDB predicate symbols, and $\{r_0, r_1, \dots\}$ to denote predicate symbols that may be either IDB or EDB one.

An *atom* is a formula written as $p(t_1, \dots, t_k)$, where p is a predicate symbol with arity $k(\geq 0)$, and the arguments $t_1, \dots, t_k$ are terms. A variable-free atom is said to be *ground*.

Let $\Sigma = \{q_1, \dots, q_s\}$ be a set of EDB predicate symbols. An *EDB fact* on Σ and $\mathbf{C}_n$ is an atom whose predicate symbol is in Σ and whose arguments are constants in $\mathbf{C}_n$. We denote the set of all EDB facts on Σ and $\mathbf{C}_n$ as $\text{EDB}(\Sigma, \mathbf{C}_n)$. An EDB instance on Σ and $\mathbf{C}_n$ is a subset of $\text{EDB}(\Sigma, \mathbf{C}_n)$, $\{q_{j_1}(\vec{c}_1), \dots, q_{j_N}(\vec{c}_N)\}$, where $\vec{c}_i$ denotes a vector of constants in $\mathbf{C}_n$.

An *initial goal* of with an IDB predicate symbol p on $\mathbf{C}_n$ is a ground atom $p(\vec{c})$, where $\vec{c}$ is a vector of constants in $\mathbf{C}_n$. A pair $(P, p(\vec{c}))$ is called a *query instance*

[1]For completeness, we define that $\mathbf{C}_0 = \varnothing$ (the empty set).

on domain $\mathbf{C}_n$, where P is an EDB instance on Σ and $\mathbf{C}_n$, and $p(\vec{c})$ is an initial goal with p on $\mathbf{C}_n$. We define the *size* of a query instance to be n (the size of the domain $\mathbf{C}_n$).

Let $k_1, \ldots, k_s$ be the arities of all predicate symbols $q_1, \ldots, q_s \in \Sigma$, respectively. A *query* Q is a countably infinite sequence

$$Q = \left\{ Q(\mathbf{C}_n) \;\middle|\; n = 0, 1, \ldots \right\}.$$

where $Q(\mathbf{C}_n)$ is a set of query instances

$$Q(\mathbf{C}_n) \subseteq \{(P, p(\vec{c})) \mid P \subseteq \mathrm{EDB}(\Sigma, \mathbf{C}_n),\ \vec{c} \in {\mathbf{C}_n}^k\}.$$

p is called the *query predicate symbol* of Q, and the vector of arities $(k_1, \ldots, k_s) \to k$ is called the *sort* of Q. When $k = 0$ (p is a zero-ary predicate symbol), Q is called a Boolean query. A query with $\Sigma = \varnothing$ (the empty set of EDB predicate symbols) is called *inherent*.

2.2. Logical query programs and the basic theorem problem. Let A be an atom and let $B_1, \ldots, B_e$ ($e \geq 0$) be (possibly zero) atoms. A formula

$$A \leftarrow B_1, \ldots, B_e$$

is called a Horn clause, as in Section 3 of Chapter II. A conjunction of atoms

$$A_1, \ldots, A_m \quad (m \geq 0)$$

is called a *goal*. When $m = 1$, "A_1" is called a unit goal.

Let $\Sigma = \{q_1, \ldots q_s\}$ be a set of EDB predicate symbols. For each $q_i \in \Sigma$ with arity k_i, we choose an IDB predicate symbol with arity k_i, denote it $p_{\neg q_i}$, and call it the *negation* of q_i. The set of negations of predicate symbols in Σ is denoted as $\bar{\Sigma}$. For simplicity, we may denote an atom composed of the negation of q_i, $p_{\neg q_i}(\vec{c})$, as $\neg q_i(\vec{c})$.

An *interpreted predicate symbol* p_* with arity $k(\geq 0)$ is an IDB predicate symbol with an associated query Q_* on Σ_* such that

$$Q_*(\mathbf{C}_n) \subseteq \{(P, p_*(\vec{c})) \mid P \subseteq \mathrm{EDB}(\Sigma_*, \mathbf{C}_n),\ \vec{c} \in {\mathbf{C}_n}^k\}.$$

Q_* is called the *interpreted rule* for p_*. In the following discussions, we adopt three inherent queries $Q_\perp$, $Q_\top$, and Q_+ such that

$$\begin{aligned}
Q_\perp(\mathbf{C}_n) &= \{\ (\varnothing,\ p_\perp(c_1))\ \} && (\textit{the minimum}),\\
Q_\top(\mathbf{C}_n) &= \{\ (\varnothing,\ p_\perp(c_n))\ \} && (\textit{the maximum}),\\
Q_+(\mathbf{C}_n) &= \{\ (\varnothing,\ p_+(c_i, c_{i+1})) \mid i \in \{1, \ldots, n-1\}\ \} && (\textit{the successor}),
\end{aligned}$$

as our interpreted rules, where $p_\perp$ and $p_\top$ are unary predicate symbols, and p_+ is a binary predicate symbol, each of which is associated with $Q_\perp$, $Q_\top$, and Q_+, respectively. We sometimes utilize descriptive notation such as

$$\begin{aligned}
X = c_1 \quad &\text{for} \quad p_\perp(X),\\
X = c_n \quad &\text{for} \quad p_\top(X),
\end{aligned}$$

and

$$Y = X + 1 \quad \text{for} \quad p_+(X, Y).$$

We denote the set of the interpreted predicate symbols as $\Gamma_{(\leqslant)}$.

Let Σ ($\subset \mathbf{P}_{\mathrm{EDB}}$) be a set of EDB predicate symbols, and Π ($\subset \mathbf{P}_{\mathrm{IDB}}$) be a set of IDB predicate symbols. We assume that Π, $\Gamma_{(\leqslant)}$ and $\bar{\Sigma}$ are disjoint. A *rule* on Σ and Π is a Horn clause composed of atoms on $\Sigma \cup \bar{\Sigma} \cup \Pi \cup \Gamma_{(\leqslant)}$, where all of the arguments are variables in $\mathbf{V}$ (none of them is instantiated by a constant), and neither an EDB predicate symbol, a negation of an EDB predicate symbol, nor an interpreted predicate symbol appears in the heads. A *basic logical query program* is a finite set of rules. We may often call it a "logical query program" or

a "program" for short. We say a program is *pure* if it has neither an interpreted predicate symbols nor a negation in the bodies.

Let P_I be a basic logical query program on Σ and Π, and let P_E be an EDB instance on Σ and $\mathbf{C}_n$. We set Σ and Π fixed, and omit mentioning them hereafter. For a given goal $N = \text{"}A_1, \ldots, A_m\text{"}$ $(m \geq 1)$, if there exist a rule $C = \text{"}A \leftarrow B_1, \ldots, B_e\text{"}$ $(k \geq 0)$ in P_I such that A and A_i is unifiable via a substitution θ for some $i \in [1, m]$, then

$$\hat{N} = (A_1, \ldots, A_{i-1}, B_1, \ldots, B_e, A_{i+1}, \ldots, A_m)\theta$$

is derived from N and C with θ. Similarly, if A_i is unifiable to a fact $q(\vec{c})$ in P_E via a ground substitution φ,

$$\widehat{N'} = (A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m)\varphi$$

is derived from N and $q(\vec{c})$ with φ. Derivation by an interpreted rule is defined as follows. Let $p_*(X_1, \ldots, X_k)$ be a term composed of an interpreted predicate symbol p_* of arity k unifiable to some A_i in N via a substitution

$$\psi = \{ \langle X_1, c_{x_1} \rangle, \ldots, \langle X_k, c_{x_k} \rangle \},$$

where all c_i's are in $\mathbf{C}_n$. If

$$(P_E \cap \mathrm{EDB}(\Sigma_*, \mathbf{C}_n),\ p_*(c_{x_1}, \ldots, c_{x_k}))$$

is in $Q_*(\mathbf{C}_n)$, the associated Boolean query on $\mathbf{C}_n$,

$$\widehat{N''} = (A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m)\psi$$

is derived from N and Q_* with substitution ψ.

Derivations, *refutations* and *refutation trees* of logical query programs are defined similarly as those of logic programs in Section 3 of Chapter II. The *basic theorem problem* of a logical query program P_I for a query instance $(P_E, p(\vec{c}))$ on $\mathbf{C}_n$ is the question,

"Does $P_I \cup P_E$ solves $p(\vec{c})$ on $\mathbf{C}_n$?"

The *base* of a program P_I on $\mathbf{C}_n$, $HB_{P_I}(\mathbf{C}_n)$, is the set of all atoms composed of predicate symbols in $\Sigma \cup \bar{\Sigma} \cup \Pi \cup \Gamma_{(\leqslant)}$ and constants in $\mathbf{C}_n$ as the arguments. The *interpretation* of a program P_I with an EDB instance P_E on $\mathbf{C}_n$ is defined as

$$I_{P_I}(\mathbf{C}_n, P_E) \stackrel{\text{def}}{=} \{A \in \mathrm{HB}_{P_I}(\mathbf{C}_n) \mid P_I \cup P_E \text{ solves } A.\}.$$

The query defined by a logical query program P_I with a query predicate symbol p of arity k is a query s.t.

$$Q_{P_I,p}(\mathbf{C}_n) \stackrel{\text{def}}{=} \{(P_E, p(\vec{c}))| \; P_E \subseteq \mathrm{EDB}(\mathbf{C}_n), \\ \vec{c} \in {\mathbf{C}_n}^k, \; p(\vec{c}) \in I_{P_I}(\mathbf{C}_n, P_E)\}.$$

We call $Q_{P_I,p}$ the *logical query* defined by P_I with p.

The basic theorem problem of P_I is translated into the membership problem of the query such that

"Is $(P_E, p(\vec{c}))$ in $Q_{P_I,p}(\mathbf{C}_n)$?"

We can view the query $Q_{P_I,p}$ as the language "recognized" by P_I with respect to p.

3. Negation, total ordering, and first-order reducibility

First we introduce complexity measures for logical query programs. The *depth* of a refutation tree is defined just in the same way as those for logic programs. The *fringe size* of a refutation is the number of leaves of the associated refutation tree of it. Note that we do not consider goal-size, since a logical query program has no structured term.

We say that a logical query program P_I is of *fringe complexity* $F(n)$ (*depth complexity* $D(n)$) if any provable query instance of size n has a refutation whose fringe-size is $< F(n)$ (depth is $< D(n)$, respectively), from $P_I \cup P_E$. A logical query

program has the *polynomial fringe property* (*superpolynomial fringe property*) if the fringe complexity of it is $n^{O(1)}$ ($2^{(\log n)^{O(1)}}$).

DEFINITION III.1. A query Q is in $\mathcal{Q}_{(\neg,\leqslant)}$-stage($D(n)$) if there is a logical query program P_I such that $Q = Q_{P_I}$ and the depth complexity of P_I is $O(D(n))$. A query Q is in $\mathcal{Q}$-stage($D(n)$) if there is a pure logical query program P_I such that $Q = Q_{P_I}$ and the depth complexity of P_I is $O(D(n))$. A query Q is in $\mathcal{Q}_{(\neg,\leqslant)}$-fringe($F(n)$) if there is a logical query program P_I such that $Q = Q_{P_I}$ and the fringe complexity of P_I is $O(F(n))$. $\mathcal{Q}$-fringe($F(n)$) is defined similarly.

The inclusion

$$\begin{aligned} \mathcal{Q}\text{-stage}(D(n)) &\subset \mathcal{Q}_{(\neg,\leqslant)}\text{-stage}(D(n)) , \\ \mathcal{Q}\text{-fringe}(F(n)) &\subset \mathcal{Q}_{(\neg,\leqslant)}\text{-fringe}(F(n)) \end{aligned}$$

is proper because of the absence of neither of negation nor total ordering [Imm87]. We must consider how much power is attatched by use of negation ($\bar{\Sigma}$) and total ordering ($\Gamma_{\leqslant}$). As with negation, it is obvious that we could never define even a simple query "p is true if and only if $r(c_1)$ is not true," by any pure logical query program. Introducing the use of negation for EDB facts only is, in a sence, a compromise point for language completeness and simplicity.

Chandra and Harel [CH84] showed that computing fixpoint queries (which can be translated to pure logical queries + arbitrary use of negation) is in $\mathcal{P}$. Conversely, Immerman [Imm86] showed that the existence of total ordering in the domain is suffice to express queries computable in polynomial time by fixpoint logic. The interpreted rules $\Gamma_{(\leqslant)} = \{p_\perp, p_\top, p_+\}$ is introduced in suggestion with the property. Indeed, when the domain is totally ordered, each of the queries can

be computed by a first-order query with order relation $<$ as

$$\begin{aligned} p_{\perp}(X) &\equiv \neg(^{\exists}W)\ [\ W < X\ ], \\ p_{\top}(X) &\equiv \neg(^{\exists}Z)\ [\ X < Z\ ], \\ p_{+}(X,Y) &\equiv X < Y\ \wedge\ \neg(^{\exists}Z)\ [\ X < Z\ \wedge\ Z < Y\ ]. \end{aligned}$$

Also note that the order relation $<$ can be trivially defined by a logical query program as

$$\begin{aligned} p_{<}(X,Y) &\leftarrow p_{+}(X,Y)\ , \\ p_{<}(X,Y) &\leftarrow p_{+}(X,Z),\ p_{<}(Z,Y)\ . \end{aligned}$$

However, it is impossible to define any of $Q_{\perp}$, $Q_{\top}$, or Q_{+} by a logical query program with an interpreted rule $<$ only. This is why we adopt $\Gamma_{(\leqslant)}$ instead of the total order relation itself.

A query Q_A is *first-order reducible* to a logical query Q_B defined by a program P_B if there exist finite set of interpreted predicate symbols $\{p_1, \ldots, p_s\}$ such that Q is defined by P_B and $\{p_1, \ldots, p_s\}$ and interpreted rules $Q_1, \ldots, Q_s$, each of which are associated with $p_1, \ldots, p_s$ respectively, are defined by first-order relational calculus [Cod70] on finite domains with total ordering. FO($\mathfrak{Q}$) denotes the class of queries first-order reducible to logical queries in $\mathfrak{Q}$.

THOEREM III.1.

$$\mathcal{Q}_{(\neg,\leqslant)}\text{-fringe}(F(n)) \ \subset\ \text{FO}(\mathcal{Q}\text{-fringe}(F(n))) \ \subset\ \mathcal{Q}_{(\neg,\leqslant)}\text{-fringe}(F(n)\, n^{O(1)}).$$

Theorem III.1 follows immediately from the next lemma.

LEMMA III.2. *For any first-order query Q, there exists a logical query program P that computes Q with fringe-size complexity $n^{O(1)}$.*

PROOF. Any formula in first-order predicate logic can be transformed into normal form such that negation symbols ($\neg$) qualify only principle formulae, using De Morgan's law.

Let f_A and f_B be formulae (in the normal form) which represent queries Q_A and Q_B respectively. Let P_A be a logical query program which defines Q_A with a predicate p_A, and P_B be a logical query program which defines Q_B with a predicate p_B. We denote the fringe-size complexity of P_A and P_B as $F_A(n)$ and $F_B(n)$ respectively.

Consider the conjunction f_A and f_B, $f_A(\vec{x}_A, \vec{y}) \wedge f_B(\vec{x}_B, \vec{y})$, where $\vec{y} = y_1, \ldots$ are the free variables shared by f_A and f_B, and $\vec{x}_A = x_A1, \ldots$ ($\vec{x}_A = x_A1, \ldots$) are the free variables that occur only in f_A (in f_B, respectively). The query represented by can be defined by the union of P_A, P_B and a rule

$$p_{A\wedge B}(\vec{X}_A, \vec{X}_B, \vec{Y}) \leftarrow p_A(\vec{X}_A, \vec{Y}),\ p_B(\vec{X}_B, \vec{Y})$$

with a predicate symbol $p_{A\wedge B}$. The fringe-size complexity of it is $F_A(n) + F_B(n)$. Similarly, the query represented by the disjunction $f_A(\vec{x}_A, \vec{y}) \vee f_B(\vec{x}_B, \vec{y})$ can be defined by the union of P_A, P_B and two rules

$$\begin{aligned} p_{A\vee B}(\vec{X}_A, \vec{X}_B, \vec{Y}) &\leftarrow p_A(\vec{X}_A, \vec{Y}), \\ p_{A\vee B}(\vec{X}_A, \vec{X}_B, \vec{Y}) &\leftarrow p_B(\vec{X}_B, \vec{Y}) \end{aligned}$$

with a predicate symbol $p_{A\vee B}$. The fringe-size complexity of it is $\max\{F_A(n), F_B(n)\}$.

The query represented by the existential bound of f_A with respect to a variable x, $\exists x[f_A(x, \vec{y})]$, can be defined by the union of P_A and a rule

$$p_{\exists A}(\vec{Y}) \leftarrow p_A(X, \vec{Y})$$

with a predicate symbol $p_{\exists A}$, and the fringe-size complexity of it is $F_A(n)$. The query represented by the universal bound of f_A with respect to a variable x,

$\forall x[f_A(x, \vec{y})]$, can be defined by the union of P_A and rules

$$\begin{aligned} p_{\forall A}(\vec{Y}) &\leftarrow X = c_n,\ p_{\forall A+}(X, \vec{Y}) , \\ p_{\forall A+}(X_1, \vec{Y}) &\leftarrow p_A(X_1, \vec{Y}),\ X_1 = X_0 + 1,\ p_{\forall A+}(X_0, \vec{Y}) , \\ p_{\forall A+}(X, \vec{Y}) &\leftarrow p_A(X, \vec{Y}),\ X = c_1 , \end{aligned}$$

with a predicate symbol $p_{\forall A}$. The fringe-size complexity of it is $n(F_A(n) + 1) + 1$.

The lemma follows by induction of composition of the formula. □

4. Alternation and logical query programs

4.1. An ATM algorithm for logical query programs. It is well known that logical query program can be computed in polynomial time of the size of input domain.

FACT III.3. *The basic theorem problem of a logical query program is in* $\mathcal{P}$.

Note that any query instance of size n can be encoded using the standard binary representation with length $n^{O(1)}$.

Fact III.3 is originally found for fixpoint queries (logical queries + arbitrary use of negation) by Chandra and Harel [CH84], although their formalization does not assume the existence of total ordering. In this section, we enter details of the complexity of computing logical queries by simulating a logical query program with a logspace bounded indexing alternating Turing machine, just like the simulation of logic programs in Section 4 of Chapter II.

Roughly speaking, as is shown in Chapter II, we can say that a logical query program is almost an ATM algorithm itself. Rules in the program corresponds to the next move relation of it. An initial goal and an EDB instance is regarded as an input string. The nondeterministic choice of a clause whose head unified with a goal corresponds to an existential branch of an ATM. The simultaneous satisfaction of the subgoals in the rule corresponds a universal branch.

The most important difference is that, conjunctive goals can share variables, while the computations of universally forked branches of an ATM must be done independently. The key idea of Shapiro's simulation [Sha84] is that the final value of a shared variable is "guessed" immediately, using existential branches, before the subgoals are forked universally. Instead of the most general unifier, a ground unifying substitution, which replace all variables in the rule into constants, is chosen.

Let P_I be a logical query program. We first show how to encode an input, viz., an initial goal $p_i(\vec{c}_0)$ and an EDB instance $P_E = \{q_{j_1}(\vec{c}_1), \ldots, q_{j_n}(\vec{c}_n)\}$ into an input tape string. Since the number of IDB predicate symbols and EDB predicate symbols occur in P_I is finite and independent from the input length, we can utilize the predicate symbols as input symbols. The constants which occur in the initial goal and the EDB facts represented in standard binary encoding.

For example, a query instance

$$(\{q_1(c_1, c_2), q_2(c_3, c_3), q_1(c_3, c_1)\},\ p_0(c_3, c_4))$$

on $\mathbf{C}_5$ is directly mapped on the input tape as

$$1\ 0\ 1\ ;\ p_0\ (\ 0\ 1\ 1\ ,\ 1\ 0\ 0\ )\ ;\ q_1\ (\ 0\ 0\ 1\ ,\ 0\ 1\ 0\ )\ ,$$
$$q_2\ (\ 0\ 1\ 1\ ,\ 0\ 1\ 1\ )\ ,\ q_1\ (\ 0\ 1\ 1\ ,\ 0\ 0\ 1\ )\ \#$$

Here $\boxed{0}$, $\boxed{1}$, $\boxed{(}$, $\boxed{)}$, $\boxed{,}$, $\boxed{;}$, $\boxed{p_0}$, $\boxed{q_1}$, $\boxed{q_2}$ and $\boxed{\#}$ are tape symbols. Since the indices of constants are encoded in $\lceil \log_2 n \rceil$ bits, the length of the input string is at most $|\Sigma| n^{k_{\max}} (= n^{O(1)})$, where $k_{\max}$ is the maximum arity of predicate symbols in Σ.

Consider the algorithm of an ATM M shown is Figure III.7. The ATM M has P_I in its finite control, and at its initial state it reads the initial goal $p_i(\vec{c}_0)$ written on the leftmost of the input tape. At each step of the simulation, it does one-step derivation according to the type of predicate symbol of the current unit goal. If

Algorithm 1

Given: a basic logical query program P_I;

Input: an integer n,

a query instance ($P_E = \{q_{j_1}(\vec{c}_1), \ldots, q_{j_N}(\vec{c}_N)\}$, $p_i(\vec{c}_0)$);

Output: whether $P_I \cup P_E$ proves $p_i(\vec{c}_0)$ on $\mathbf{C}_n$ or not;

function ASK($q(\vec{c})$: q is an EDB predicate symbol, $\vec{c}$ is a constant vector;): Boolean;

begin

if $q(\vec{c}) \in P_E$ **then** ASK := true **else** ASK := false;

end;

procedure DERIVE($r(\vec{c})$: a unit goal);

begin

if $\exists q \in \Sigma\ [r = q]$ **then**

begin if not ASK($r(\vec{c})$) **then reject end**

else if $\exists p_{\neg q} \in \vec{\Sigma}\ [r = p_{\neg q}]$ **then**

begin if ASK($q(\vec{c})$) **then reject end**

else if $\exists p_* \in \Gamma_{(\leqslant)}\ [r = p_*]$ **then**

begin if $(\varnothing, p_*(\vec{c})) \notin Q_*(\mathbf{C}_n)$ **then reject end**

else

begin

Choose a rule "$A \leftarrow B_1, \ldots, B_k$" in P_I; (existentially)

Guess a ground substitution θ; (existentially)

if $A\theta \neq r(\vec{c})$ **then reject**;

for $\forall i \in \{1, \ldots, k\}$ **do** (universally)

DERIVE($B_i\theta$);

end;

end;

begin {*of the main routine*}

Say "YES" **if** DERIVE($p_i(\vec{c}_0)$) is successfully done;

end.

FIGURE III.7. An ATM algorithm simulating logical query programs.

it is an EDB predicate symbol in Σ or negation of an EDB predicate symbol, it reads the input tape and checks whether the current goal is in the EDB instance. If it is an interpreted predicate symbol (say p_*), M interprets Q_* for the given arguments. If it is an IDB predicate symbol in Π, it existentially chooses a rule in P_I, and writes on its work tape a substitution θ. Next it computes $A\theta$, verifies if $A\theta = r(\vec{c})$, and erase the current goal. Then it universally chooses all subgoals B_i's in the rule and recursively dose the same for the new goal $B_i\theta$ for each i.

During the simulation, each goal is kept as a string, whose length is at most $O(\log n)$, on the work tape. Since the number of variables appear in a rule is at most constant, the substitution θ can be encoded in a string of length $O(\log n)$. Thus the ATM M uses space $O(\log n)$. Each call of DERIVE is done in time $O(\log n)$ and each call of ASK is done in time $O(\log n)$. Each evaluation of an interpreted rule is done in time $(\log n)^{O(1)}$ with tree-size $n^{O(1)}$. For a derivation tree of depth D and fringe-size F, the corresponding simulation is done in time $O(D \log n)$, with space $O(\log n)$ and tree-size $O(DF \log n)$. We now have:

THOEREM III.4. *For every logical query program P_I of depth complexty $D(n)$ and fringe complexity $F(n)$, there exists an indexing ATM M of depth complexity $O(D(n) \log n)$, space complexity $O(\log n)$ and tree-size complexity $O(D(n)F(n) \log n)$, , such that M accepts a query instance iff the union of the EDB instance and P_I solves the initial goal on the domain.*

4.2. Programs simulating ATMs. Let us show the converse of Theorem III.4, i.e., how a logspace bounded alternation is simulated by a logical query program.

One problem on simulating an automaton with a *pure* logical query programs is representation of the input string as an EDB instance. Since an EDB instance itself is an unordered set of facts, there seems no direct way to express naturally the symbol at each position on the input tape without using the total ordering.

For example, if one prepared an EDB instance on $\mathbf{C}_n$,

$$\{q_{j_1}(c_1),\ q_{j_2}(c_2),\ \ldots, q_{j_n}(c_n)\}$$

for an input string of length n, "$q_{j_1}q_{j_2}\ldots q_{j_n}$", the program would have no way to know that thre position represented by 1 and the position represented by 2 is contiguous. Instead of this, Ullman et al. defined a class of EDB instances of the form

$$\{q_{j_1}(c_1, c_2),\ q_{j_2}(c_2, c_3),\ \ldots, q_{j_n}(c_n, c_{n+1})\}$$

and restricted EDB instances in such a class. However a program which has the polynomial fringe property for the restrict EDB instances does not necessarily have the polynomial (or even superpolynomial) fringe property for arbitrary EDB instances. In contrast, no such problem arises in our formalization, since we assume the total ordering as interpreted rules. If the character at the ith position of an input string is q_{j_i}, it is directly represented as a fact $q_{j_i}(c_i)$. This brings language completeness to logical query programs; as we show here, any logspace bounded alternating Turing machine has strict correspondence to a logical query program whose fringe-size complexity is the same as the tree-size complexity of the ATM up to a polynomial factor.

Instead of logspace bounded ATMs, we utilize *alternating multihead finite automata.* An alternating multihead finite automaton is a variant of a multihead finite automaton [Kas87a] with the power of *alternation.*

LEMMA III.5. *For any logspace bounded ATM M, there exist an alternating multihead finite automaton M' such that M' accepts a string w with tree size $Z(n)n^{O(1)}$ iff M accepts a string w with tree size $Z(n)$.*

Remark. See the simulation of a logspace bounded DTM by a deterministic multihead finite automaton [Kas87a]. □

Let M be an alternating k-head finite automaton. Without loss of generality, we may assume that the tape symbols for the input tape is $\{0,1\}$, and that the number of branches on every universal state is just 2. We set the set of EDB predicate symbols to be $\{q\}$, where q is a predicate symbol symbol with arity 1, and the set of query predicate symbols to be $\{p_0\}$, where p_0 is a predicate symbol symbol with arity 0. Encoding of the input string of length n,

$$w = \text{“}x_1x_2\ldots x_n\text{”} \quad x_i \in \{0,1\}$$

to an query instance on $\mathbf{C}_n$ is given as (P_E, p_0) s.t.

$$P_E(w) = \{p(c_i) \mid i \in \{1,\ldots,n\},\ x_i = 1\}.$$

Let $\mathcal{A}_w$ be the set of all configurations of M for a given input string w. A configuration of M, say α, is specified by a k tuple of the current state $s_{(\alpha)}$, and the position of the first to the kth heads $h_0,\ldots,h_k$. We denote the vector of constants representing the head positions of α as

$$\vec{c}_\alpha \stackrel{\text{def}}{=} (c_{h_1},\ldots,c_{h_k}).$$

We define a logical query program P_M such that there exist a realizable computation of M on an input string w of length n from a configuration α iff $P_E(w) \cup P_M$ proves $p_{s_{(\alpha)}}(\vec{c}_\alpha)$ on $\mathbf{C}_n$.

First we define rules for one-step transitivity of two configurations. Let

$$p^{(1)}_{s_{(\alpha)}\to s_{(\beta)}}(\vec{c}_\alpha, \vec{c}_\beta)$$

be a clause, which represents that there is a one-step transition from α to β. For simplicity of description, we may assume that $k = 3$. For example, assume that there is a one-step transition from a state s_A to a state s_B with head movement of the first head left, the second head right and the third head remained, when the input value of each heads is 1, 0, 1, The corresponding rule for this transition is

written as

$$p^{(1)}_{s_A \to s_B}(X_1, X_2, X_3,\ Y_1, Y_2, X_3) \ \leftarrow$$
$$q(X_1),\ \neg q(X_2),\ q(X_3),\ X_1 = Y_1 + 1,\ Y_2 = X_1 + 1\ .$$

Define $P^{(1)}_M$ be the set of all rules defined similarly for all one-step transition rules of M.

Next we define the following rules for each state of M. For each univarsal state $s_\wedge$ whose next states are s_A and s_B, generate a rule

$$p_{s_\wedge}(\vec{X}) \ \leftarrow p^{(1)}_{s_\wedge \to s_A}(\vec{X}, \vec{Y}),\ p^{(1)}_{s_\wedge \to s_B}(\vec{X}, \vec{Z}),\ p_{s_A}(\vec{Y}),\ p_{s_B}(\vec{Z}),\ \vec{Y} \neq \vec{Z}\ .$$

For each existential state $s_\vee$ and each of the next states of $s_\vee$, say s_A, generate a rule

$$p_{s_\vee}(\vec{X}) \ \leftarrow p^{(1)}_{s_\vee \to s_A}(\vec{X}, \vec{Y}),\ p_{s_A}(\vec{Y}).$$

For each accepting state s_{accept}, generate a rule

$$p_{s_{\text{accept}}}(\vec{X}) \ \leftarrow.$$

Define $P^{(+)}_M$ be the set of all rules defined for all states of M. The program P_M is the union of $P^{(1)}_M$ and $P^{(+)}_M$.

Initial goal is set as $p_{s_0}(\vec{c}_0)$, where s_0 is the initial state andp $\vec{c}_0$ is the vector of integers which represents the initial head positions $1, \ldots, 1$ of M. Consider a derivation of $p_{s_0}(\vec{c}_0)$ with $P_M \cup P_E(w)$. Obviously, each derivation is performed for exactly one move of M. Thus, a refutation of $p(\vec{c}_0)$ is constructible iff an acceptable computation of M on w exists. If the tree-size of the computation is $Z(n)$, the fringe-size of the refutation is $O(Z(n))$.

THOEREM III.6. *For any logspace bounded ATM M, there exists a logical query program P_M, such that M accepts an input string w of length n with tree-size $Z(n)$ if and only if $P_M \cup P_E(w)$ solves $p(\vec{c}_0)$ with fringe-size $Z(n) \cdot n^{O(1)}$.*

Remark. The theorem follows immediately from Lemma III.5 and the discussion above. □

Summarizing the two theorem shown in this section, we get:

THOEREM III.7. *For any tape-constructible function* $D(n) = \Omega(\log n)$,

$$\begin{aligned} \mathcal{Q}_{(\neg,\leqslant)}\text{-stage}(D(n)) &= \mathcal{Q}_{(\neg,\leqslant)}\text{-fringe}(2^{O(D(n))}) \\ &= A\text{-}SpSz(\log n, 2^{O(D(n))}) \ . \end{aligned}$$

5. Parallelizing logical query programs

There holds an obvious inclusion that

$$\mathcal{Q}_{(\neg,\leqslant)}\text{-stage}(D(n)) \subset \mathcal{Q}_{(\neg,\leqslant)}\text{-fringe}(2^{O(D(n))}),$$

since any proof with depth $D(n)$ can have at most $2^{O(D(n))}$ fringes. In this section, we show that in fact

$$\mathcal{Q}_{(\neg,\leqslant)}\text{-stage}(D(n)) = \mathcal{Q}_{(\neg,\leqslant)}\text{-fringe}(2^{O(D(n))})$$

holds for any tape-constructible function $D(n) = \Omega(\log n)$. This indicates that any logical query program with fringe-size complexity $F(n)$ can be *parallelized* as the equivalent program of depth complexity $O(\log F(n))$.

We present a simple syntactic transformation of a program P_I into an equivalent parallelization $\tilde{P}_I$. Here we say that a program $\tilde{P}_I$ is a *parallelization* of P_I if $P_I \subset \tilde{P}_I$. The additional rules are generated by the following procedure.

Initialization: Set $\hat{P}_I = \varnothing$.

Iterative rules: We generate two kinds of rules, say *activation rules* and *pebbling rules*, for each IDB rule in P_I. For example, let P_I has a rule

$$p_0(\vec{X}_0) \leftarrow p_1(\vec{X}_1),\ p_2(\vec{X}_2),\ q(\vec{Y}),\ r(\vec{Z}) \ ,$$

where p_0, p_1 and p_2 are IDB predicate symbols, q is an EDB predicate symbol and r is an interpreted predicate symbol. Then append the following rules to $\hat{P}_I$.

- *activation rules*

$$\Psi_{p_0 \leftarrow p_1}(\vec{X}_0, \vec{X}_1) \leftarrow p_2(\vec{X}_2),\ q(\vec{Y}),\ r(\vec{Z})$$

  and the corresponding rule for $\Psi_{p_0 \leftarrow p_2}$.

- *pebbling rules*

$$p_0(\vec{X}_0) \leftarrow \Psi_{p_0 \leftarrow p_1}(\vec{X}_0, \vec{X}_1), p_1(\vec{X}_1)$$

  and the corresponding rule for $\Psi_{p_0 \leftarrow p_2}$.

Do this for all rules in P_I.

Transitive closure: Let p_i, p_j and p_k be (not necessarily distinct) IDB predicate symbols occur in some rule heads in P_I. Append the next rule

$$\Psi_{p_i \leftarrow p_j}(\vec{X}_i, \vec{X}_j) \leftarrow \Psi_{p_i \leftarrow p_k}(\vec{X}_i, \vec{X}_k), \Psi_{p_k \leftarrow p_j}(\vec{X}_k, \vec{X}_j)$$

to $\hat{P}_I$. Do this for all triplet of IDB predicate symbols in P_I.

The resulting parallelized program is $\tilde{P}_I = P_I \cup \hat{P}_I$. The frige-size complexity of the parallelized program is $O((F(n))^2)$, where $F(n)$ is the fringe-size complexity of the original program.

PROPOSITION III.8. *If P_I is a logical query program with fringe-size complexity $F(n)$, then its parallelized equivalent $\tilde{P}_I$ is of depth complexity $O(\log F(n))$.*

The proposition is proved by Ullman and Van Gelder [UV88] as a PRAM algorithm for the basic theorem problem (see also [Ruz80]). Kanellakis presented it as a program transformation [Kan88], although he did not show any concrete procedure.

We can say that any program with the polynomial fringe property can be parallelized so that the depth complexity of it be $O(\log n)$.

COROLLARY III.9.

$$\begin{aligned} \mathcal{Q}_{(\neg,\leqslant)}\text{-stage}(D(n)) &= \mathcal{Q}_{(\neg,\leqslant)}\text{-fringe}(2^{O(D(n))}), \\ \mathcal{Q}\text{-stage}(D(n)) &= \mathcal{Q}\text{-fringe}(2^{O(D(n))}). \end{aligned}$$

Note that the procedure of parallelization does not assume existence of neither negation nor total ordering.

EXAMPLE III.1. Let us consider a logical query program P consists of rules

$$\begin{aligned} p(X,Y) &\leftarrow p(X,Y_1),\ X_1 = Y_1 + 1,\ p(X_1,Y)\ , \\ p(X,Y) &\leftarrow q(X),\ \neg q(Y),\ p(X_1,Y_1),\ X_1 = X + 1,\ X = Y_1 + 1\ , \\ p(X,Y) &\leftarrow X = Y + 1\ , \end{aligned}$$

where q is an EDB predicate symbol and p is an IDB predicate symbol. An example of a derivation tree of an EDB instance

$$(\ \{q(c_1),\ q(c_2),\ q(c_3),\ q(c_7),\ q(c_9),\ \},\ p(c_1,c_{10})\ )$$

on $\mathbf{C}_{10}$ is shown in Figure III.8 (a). (We omit showing leaf nodes, i.e., derivation by interpreted predicates, EDB facts, or negation of EDB favts. We abbreviate atoms, e.g., $p(c_1,c_{10})$, as "p(1,10)".)

The additional rules for parallelization are

$$\begin{aligned} \Psi_{p\leftarrow p}(X,Y,\ X,Y_1) &\leftarrow X_1 = Y_1 + 1,\ p(X_1,Y) && (\textit{activation}), \\ \Psi_{p\leftarrow p}(X,Y,\ X_1,Y) &\leftarrow X_1 = Y_1 + 1,\ p(X,Y_1) && (\textit{activation}), \\ \Psi_{p\leftarrow p}(X,Y,\ X_1,Y_1) &\leftarrow q(X),\ \neg q(Y), && \\ &\quad X_1 = X + 1,\ X = Y_1 + 1 && (\textit{activation}), \\ p(X,Y) &\leftarrow \Psi_{p\leftarrow p}(X,Y,\ X_1,Y_1),\ p(X_1,Y_1) && (\textit{pebbling}), \\ \Psi_{p\leftarrow p}(X_0,Y_0,\ X_1,Y_1) &\leftarrow \Psi_{p\leftarrow p}(X_0,Y_0,\ X_2,Y_2), && \\ &\quad \Psi_{p\leftarrow p}(X_2,Y_2,\ X_1,Y_1) && (\textit{transitive closure}). \end{aligned}$$

The parallelized proof is shown in Figure III.8 (b).

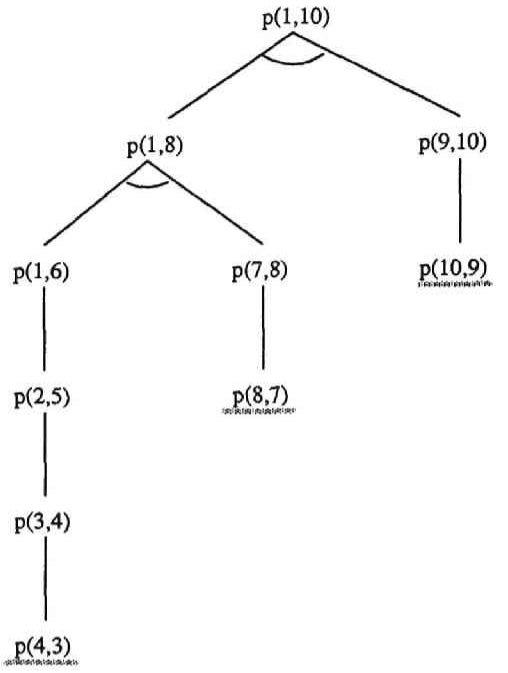
Kanellakis also showed that there exists a program P_I such that $Q_{P_I} \in \mathcal{Q}$-stage$(\log n)$ but the fringe-size complexity of P_I is $(\log n)^{\omega(1)}$ (more than superpolynomial of n). This suggests that not only the depth complexity but also the fringe-size complexity is an unstable measure of the complexity of the query itself. They represent properties of the program itself, not those of the query defined by it. This sounds of course reasonable, since one could write a bad program with as much redundancy as possible. All we can say is that "Only good programs can effectively be parallelized." So we may ask "How can we know a program is good or not?" However, a pessimistic result has been proved by Ullman and Van Gelder [UV88]

PROPOSITION III.10. *It is undecidable whether an elementary chain rule program has polynomial fringe-size complexity.*

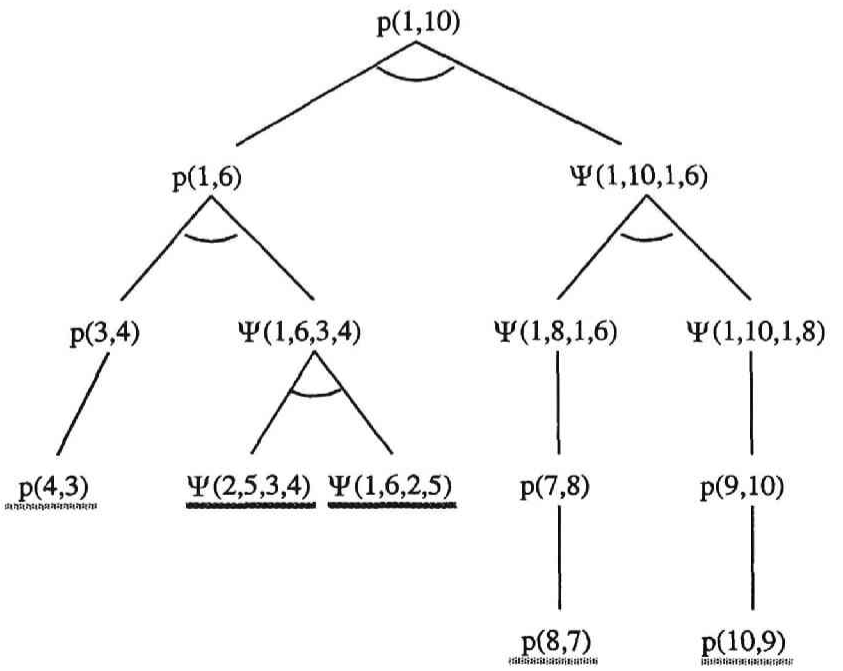
The proposition indicates that there exist no general procedure to determine whether an arbitrary logical query program can be effectively parallelized or not. In the next section we will show that some syntactic restrictions provide a good programming guide to writing easily-parallelized programs which do not necessarily have enough explicit parallelism itself.

6. Linear programs and non-confluent programs

A *linear* logical query program is one in which each rule has at most one IDB subgoal (the rest subgoals are either with EDB predicate symbols, negation, or interpreted predicate symbols). It has been discovered independently by many researchers that the basic theorem problem of linear programs is in $\mathcal{NC}$ (see [UV88]). The fact that it is in $\mathcal{NL}$ is proved by Immerman [Imm87]. He also showed the converse, i.e. any query in $\mathcal{NL}$ can be defined by a linear program. This was



(a) An example program.



(b) Parallelization of (a).

FIGURE III.8. Parallelization of a logical query program.

shown as a relation between $\mathcal{NL}$ and first-order queries with transitive closure operation under the existence of total ordering [Imm87].

PROPOSITION III.11. *The set of all queries defined by linear programs is exactly equal to $\mathcal{NL}$.*

Problems in $\mathcal{NL}$ are guaranteed to have potential parallelism, since any linear program has the polynomial fringe property. However, there exist so many problems in $\mathcal{NC}$ that are convinced not to be in $\mathcal{NL}$. One of the most typical example of such problem is CFL recognition.

We define a more general class of "good" logical query programs. A logical query program is called *non-confluent* if there exist a refutation tree in which all recursive subgoals are distinct for any query instance solvable via the program.

LEMMA III.12. *Any non-confluent program has the polynomial fringe property.*

PROOF. Each subgoal are distinguished by its predicate symbol symbol and constants as its argument. Thus for a domain of size n, the number of admissible subgoals are at most $|\mathbf{P}| \cdot n^{k_{\max}}$, where $k_{\max}$ is the maximum of the arities of the predicate symbol symbols. □

Next we show that most of programs which is known to have the polynomial fringe property are non-confluent. The first example is the class of linear programs.

THOEREM III.13. *Any linear logical query program is non-confluent.*

PROOF. If a refutation tree of a linear program (approximately a path) has two identical subgoals. Then the derivation tree given by omitting all derivations between the two subgoals, which form a redundant cycle, is also a refutation. □

Ullman and van Gelder [UV88] investigated elementary chain-rule programs with the polynomial fringe property in association with context-free grammars. See [UV88] for the terminology used here.

Let P_I be a elementary chain rule program with IDB predicate symbols p, $p_1, p_2, \ldots$, and EDB predicate symbols, $q_1, q_2, \ldots$, all of which p properly depends upon. Let G be the CFG obtained by considering p as the start symbol, considering $p_1, p_2, \ldots$ as the nonterminals and considering $q_1, q_2, \ldots$ as the terminals. The productions of G correspond to the rules of P_I in the way such that the right side production corresponds to the chain of subgoals of a rule. Let L be the language generated by G. Let D_1 be the Dyck language on one kind of parentheses.

PROPOSITION III.14 (GSM MAPPING THEOREM). *If L is the GSM mapping of $D_1\$$ for some GSM M, then P_I with p has the polynomial fringe property.*

Some programs which are proved to have polynomial fringe property by GSM mapping theorem are shown in [UV88]. We relate the theorem to the non-confluence.

THOEREM III.15. *If a program satisfies the assumption of the GSM mapping theorem, then it can be transformed into non-confluent form.*

Remark. In the proof of the GSM mapping theorem in [UV88], "depth" and a triple ID(i) plays an important role to count the number of fringes in a refutation. Specifically, the following two claims holds.

(1) For $1 \leq i < j \leq r$, if $d_i = d_j$ (depth is the same) then $\mathrm{ID}(i) \neq \mathrm{ID}(j)$.

(2) The maximum depth is polynomially bounded.

We can modify the original program P_I to compute auxiliarily the associated depth and ID(i). Then the subgoals in a refutation is distinguished by either the depth or the ID. □

Next we show that non-confluent programs have enough power to describe all queries in $\mathcal{Q}_{(\neg,\leqslant)}$-stage($\log n$).

Let $G = (\Gamma, \Sigma, P, S)$ be a context-free grammar in Chomsky normal form [Kas87b]. Here Γ is the set of nonterminal symbols, Σ is the set of terminal symbols, P is the set of production rules, and S is the start symbol. A terminal symbol in Σ (say a) corresponds to a unary EDB predicate symbol (denoted q_a), and a nonterminal symbol in Γ (say A, for example) corresponds to a binary IDB predicate symbol (denoted p_A). The IDB predicate symbol for the start symbol S, p_S, is the unique external IDB predicate symbol. A string of length n on Σ^*, $w = a_1 a_2 \ldots a_n$ corresponds to a EDB instance

$$P_E(w) = \{\ q_{a_1}(c_1), q_{a_2}(c_2),\ \ldots,\ q_{a_n}(c_n)\ \}.$$

The query defined by G is

$$\{(n, P_E(w), p_S(c_1, c_n)) \mid w \in \Sigma^*,\ S \stackrel{*}{\underset{G}{\Longrightarrow}} w\}.$$

Let us show a program simulating G, say P_G. For each non-terminal production $A_0 \rightarrow A_1 A_2$ we create a rule

$$p_{A_0}(X, Y) \ \leftarrow\ p_{A_1}(X, Z),\ W = Z + 1,\ p_{A_2}(W, Y)$$

and register it to P_G. Each terminal production $A \rightarrow a$ becomes

$$p_A(X, X) \ \leftarrow\ q_a(X), \neg q_{b_1}(X), \ldots, \neg q_{b_{|\Sigma|-1}}(X)\ ,$$

where $b_1, \ldots, b_{|\Sigma|-1}$ are all of the symbols in $\Sigma - \{a\}$. Obviously a refutation of P_G on $P_E(w)$ is related to the corresponding production of $S \stackrel{*}{\underset{G}{\Longrightarrow}} w$.

THOEREM III.16. *The logical query program defined by a context-free grammar in Chomsky normal form is non-confluent.*

PROOF. When a rule

$$p_{A_0}(X, Y) \ \leftarrow\ p_{A_1}(X, Z),\ W = Z + 1,\ p_{A_2}(W, Y)$$

is applied to a goal $p_{A_0}(c_x, c_y)$, the associated territories of the goal and the two new subgoals are $[x, y]$, $[x, z]$ and $[z + 1, y]$ respectively. Here proper inclusion $[x, z] \subsetneq [x, y]$, $[z + 1, y] \subsetneq [x, y]$ and disjointness $[x, z] \cap [z + 1, y] = \varnothing$ hold. The inclusion and disjointness is inherited to all descendants of the refutation tree. Thus no two distinct goals in a refutation tree are instantiated by the same arguments. □

As shown in Theorem III.16, there exist a non-confluent logical query program which exactly computes the recognition problem of the CFL defined by any context-free grammar. Also note that any query logspace reducible to a CFL can be computable by a logical query program, since the logspace transducer can be emulated by a linear program (with first-order queries as interpreted predicate symbols). The next theorem follows immediately.

THOEREM III.17. *The set of all queries defined by non-confluent programs is equal to* LOGCFL

Noting that LOGCFL = A-$SpSz(\log n, n^{O(1)})$ [Ruz80], we get:

COROLLARY III.18. *For any logical query defined by a logical query program with the polynomial fringe property, there exists a non-confluent program that computes the query.*

PROOF. From Theorem III.7, $\mathcal{Q}_{(\neg,\leqslant)}$-fringe$(n^{O(1)}) = A$-$SpSz(\log n, n^{O(1)})$ holds. □

Note that we can choose the program so that its depth complexity is $O(\log n)$, since the parallelizing procedure shown in Section 5 does not affect non-confluence of a program.

Classification of logical query programs by the depth complexity (or equivalently by the fringe-size complexity) is characterized via the hierarchy of subclasses of $\mathcal{P}$.

COROLLARY III.19. *The following classification of queries defined by logical queries with respect to depth complexity holds.*

$$\begin{aligned} \mathcal{Q}_{(\neg,\leqslant)} &= \mathcal{Q}_{(\neg,\leqslant)}\text{-stage}(n^{O(1)}) &&= \mathcal{P} \\ &\quad \mathcal{Q}_{(\neg,\leqslant)}\text{-stage}(\log^{O(1)} n) &&= \mathcal{NC} \\ \mathcal{Q}_{(\neg,\leqslant)}\text{-non-confluent} &= \mathcal{Q}_{(\neg,\leqslant)}\text{-stage}(\log n) &&= \text{LOGCFL} \\ \mathcal{Q}_{(\neg,\leqslant)}\text{-linear} &= &&= \mathcal{NL} \end{aligned}$$

Here $\mathcal{Q}_{(\neg,\leqslant)}$-linear, $\mathcal{Q}_{(\neg,\leqslant)}$-non-confluent, *and* $\mathcal{Q}_{(\neg,\leqslant)}$ *are queries defined by linear programs, non-confluent programs, and arbitrary logical query programs respectively.*

We can apply many results on the theory of computational complexity, obtained on conventional models, to logical query programs. For example, the negation of any linear program is also computable by a linear program, by applying Immerman's theorem [Imm88, Sze87]:

PROPOSITION III.20 (IMMERMAN). $\mathcal{NL}$ *is closed under complementation.*

COROLLARY III.21. *For any linear logical query program* P_I, *there exists a linear logical query program* $\bar{P}_I$ *such that an initial goal* $p_I(\vec{c})$ *is not provable from* P_I *and* P_E, *iff* $\bar{p}_I(\vec{c})$ *is provable from* $\bar{P}_I$ *and* P_E.

Similarly, the negation of every non-confluent logical query program is computable by a non-confluent logical query program.

THOEREM III.22. *For any non-confluent logical query program* P_I, *there exists a non-confluent logical query program* $\bar{P}_I$ *psuch that an initial goal* $p_I(\vec{c})$ *is not provable from* P_I *and* P_E, *iff* $\bar{p}_I(\vec{c})$ *is provable from* $\bar{P}_I$ *and* P_E.

This follows from the fact that LOGCFL is closed under complementation [BCD+87].

Ullman et al. proved $\mathcal{P}$-completeness of the basic theorem problems for some logical query programs, by reduction from the Monotone Circuit Value Problem [UV88]. Similar discussion for linear programs leads $\mathcal{NL}$-completeness of a linear logical query program by reduction from the Graph Accessibility Problem [Imm87]. As an additional remark, we mention a LOGCFL-complete logical query program by reduction from the word problem of the hardest context-free language [Gre73].

COROLLARY III.23. *The basic theorem problem of the logical query program defined by the hardest context-free language is $\mathcal{NC}^1$-complete for* LOGCFL.

7. Concluding remarks

We have shown a new formalization of logical query languages as a theoretical model of computation, and have shown a relationship between logical query programs and tree-size bounded alternation. We present a class of effectively parallelizable logical queries, which is defined by a non-confluent programs, and showed that it exactly is LOGCFL.

Non-confluency of a program is, however, not so clear as linearity; there is no procedure determining whether a program is non-confluent or not, although we can mostly write a "explicitly" non-confluent program for a given query in LOGCFL. There may be a better substitution which can be trivially determined by the syntax of a program and still have power to express all queries in CFL. Afrati and Papadimitriou [AP93] showed that any query defined by a simple chain rule program is either in LOGCFL or $\mathcal{P}$-complete. But it is still open if there is some syntactical property which exactly characterizes the queries in $\mathcal{NC}$ (logical queries with the superpolynomial fringe property).

CHAPTER IV

Optimal Data Transfer on Bus-Connected Parallel Machines

1. Introduction

Memory hierarchy is one of the most important subjects in computer science, and not a few studies, some are theoretical and others are practical, have been done. We are concerned in theoretical aspects of this subject here. A practical approach is given in the next chapter.

A pioneering theoretical work was done by Floyd [Flo72]. He established a simple computation model, composed of a processor with internal memory and a disk. Records are transferred blockwise between the disk and the internal memory, and only two blocks of records can reside on the memory at a time. He showed lower bounds of data transfers required for permuting and related problems. The lower bound for permuting is generalized by Tsuda et. al. [TST83] on models which have practical size of internal memory. Problems other than permuting, viz., sorting, FFT, etc., are also investigated [HK81, SV85]. and some lower bounds and optimal algorithms for these problems are shown by Aggarwal and Vitter [AV88]. The results are generalized on models with multiple disks which have parallel data-transfer facility [VS90].

In this chapter, we propose a new parallel computation model, which is a multiprocessor extension of the conventional two-level memory model. It is composed

of processors with internal memory and a disk as a file server, which are connected by a shared I/O bus. Computation time is measured by the number of I/O operations (data-transfer operations via the bus). Lower bounds of required data transfers and optimal algorithms on it are discussed.

The power of a model essentially depends on whether *broadcasting* to the bus is allowed or not. We show tight lower and upper bounds of the number of required data transfer operations for sorting, permutation FFT, matrix transposition and matrix multiplication, on both of models with and without broadcasting respectively. When broadcasting is permitted, sorting, permutation and matrix transposition can be done on the p-processor model $\Theta(\log p)$ times faster than on a uniprocessor model. FFT is accelerated by $\Theta(\log \min\{p, B\})$ times, where B is the block size, and matrix multiplication is accelerated by $\sqrt{p}$ times. In contrast, when broadcasting is not available, at most constant-factor acceleration can be attained for all problems discussed here.

Our results are multiprocessor extension of the bounds on the conventional model of two-level storage [AV88]. The proposed model reflects not only parallelism of operations in multiprocessing, but also parallelism of block data transfer and broadcasting. This enables to distinguish, for example, sorting and FFT with respect to parallel data-transfer complexity, while the two has the same complexity on uniprocessor two-level memories.

In Section 2, we will give the definition of bus-connected parallel two-level memories as our computation model. The lower bounds and optimal algorithm for sorting, permuting, FFT, matrix transposition and matrix multiplication will be presented in Section 3, Section 4, Section 5, Section 6, and Section 7, respectively.

2. Computation models

The computation model employed in this chapter is composed of p processors and a disk (file server) connected by a single shared bus, as shown in Figure IV.9. Each processor has its own local memory of size M. Data is transferred among the processors and the disk in blocks of B records.

Our parameters are

- N: # of records to sort (*file size*);
- M: # of records that can fit into internal memory (*memory size*);
- B: # of record that can be transferred in a single block (*block size*);
- p: # of processors

where $1 \leq B$ and $2B \leq M < N$. Cells on internal memories and on the disk are linearly numbered by an index; The address of a cell on the internal memory of the ith processor are in $[iM, (i+1)M - 1]$ $(i \in \{0, \ldots, p-1\})$ and the address of any cell on the disk are in $[pM, +\infty)$. We assume that the size of each cell is not smaller than $\lfloor 1 + \log_2 N \rfloor$ bits, so as to store the address of an input record in it.

At a data-transfer phase, either a processor or the file server can transmit a block data to the bus. When a processor transmits a block, it chooses any B cells on its memory, packs data in them into a block at any order, and sends it. When the file server transmits a block, it sends a block in any sector on the disk without any modification or reordering of it. On a model with *broadcasting*, all processors and the file server can receive the data being transmitted on the bus simultaneously. Either of, what is called, broadcasting or multicasting is available on it. In contrast, on a model without broadcasting, only one among the file server and the processors can receive the data on the bus. On either model, a processor can receive the data as a whole or selectively, and put them at any cells on the internal memory. The file server puts the received block at any sector on the disk

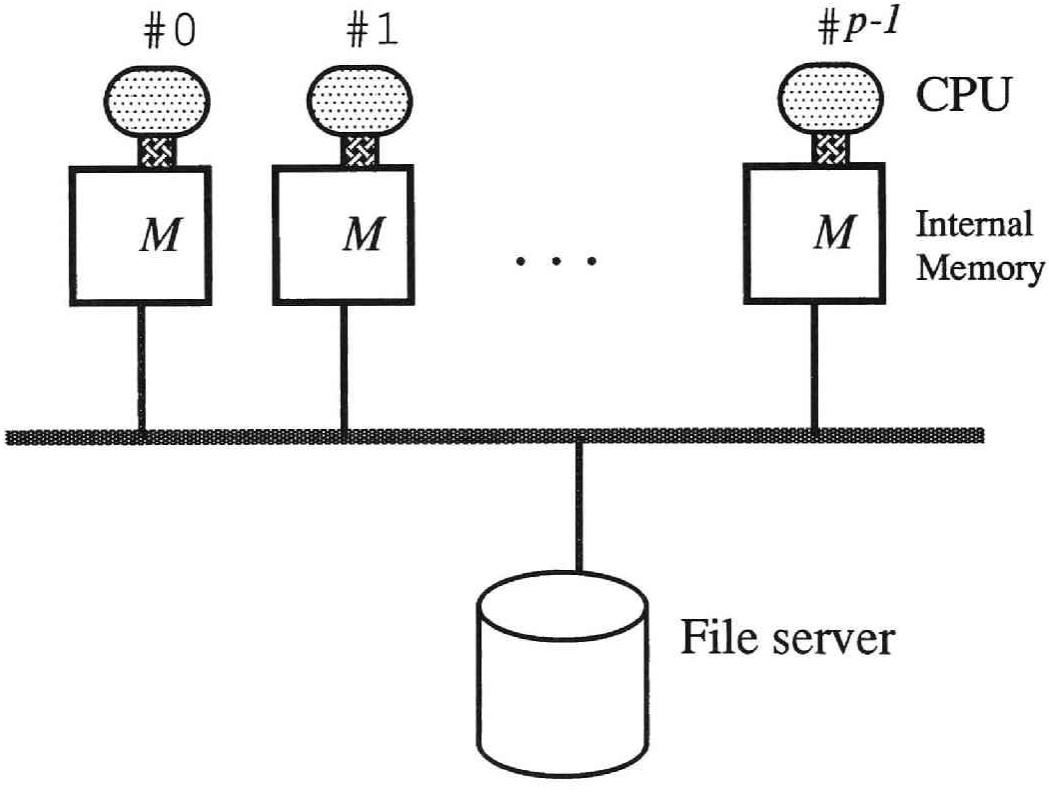


FIGURE IV.9. A bus-connected parallel two-level memory.

without any modification of the contents.

Computation time on these models is measured by the number of data-transfer phases. Each processor is assumed to have so much computation power that we may neglect the time required for operations on its internal memory. We refer our model as the *bus-connected parallel two-level memory model*, or the *parallel two-level memory* for short.

The bus-connected parallel two-level memory can be regard as a multiprocessor extension of the well-known two-level memory [SV85],[AV88]. We must also notice that a bus-connected parallel two-level memory with p processors each of which has internal memory of size M can be emulated by a two-level memory with internal memory of size pM, since the number of processor contributes only the size of internal memory and does not enhance the power of intraprocessor operations. Hierarchy of the models are shown in Figure IV.10. This may be considered as the relation among a uniprocessor system, a distributed memory multiprocessor system and a shared memory multiprocessor system.

3. Sorting

3.1. Problem definitions. The problem definition of sorting N records is stated as:

Problem Instance: The internal memory of each processor is empty, and N records reside on the contiguous N/B sectors at the beginning of the disk of the file server.

Internal Operations: Each processor compares two records and moves of a record on the internal memory into some cell of the memory.

Goal: The internal memory of each processor is empty, and the N records reside in sorted nondecreasing order on the contiguous N/B sectors at the beginning of the disk of the file server.

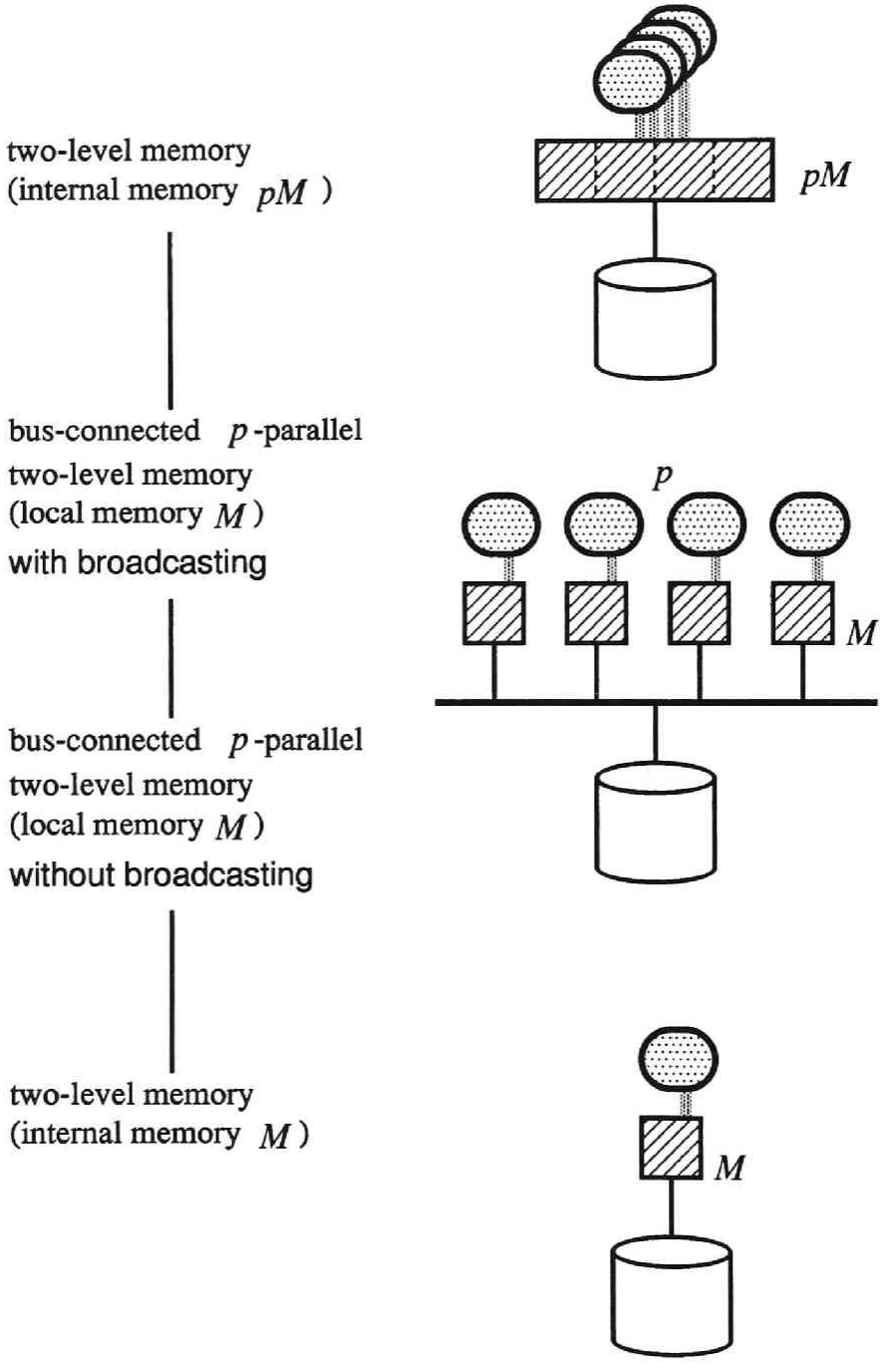


FIGURE IV.10. Comparison of two-level memory models.

3.2. The lower bound. A known result on the data-transfer complexity of sorting on the (uniprocessor) two-level memory model is as follows [AV88].

PROPOSITION IV.1 (AGGARWAL, VITTER 1988). *The average-case and worst-case number of I/Os (data transfers between the disk and the processor) required for sorting N records on a two-level memory is* $\Theta\left(\frac{N}{B}\frac{\log(N/B)}{\log(M/B)}\right)$.

The relation between a two-level memory and a parallel two-level memory shown in the previous section immediately leads the following lower bound.

COROLLARY IV.2. *The number data transfers required for sorting N records on a bus-connected two-level memory is* $\Omega\left(\frac{N}{B}\frac{\log(N/B)}{\log(pM/B)}\right)$.

Corollary IV.2 holds on either model with or without broadcasting. Further, we improve the lower bound on models without broadcasting,

For simplicity, we pose two additional restrictions to our model.

- Data transfer is done as either *input* (from the disk of the file server to the internal memory of a processor) or *output* (from the internal memory to the disk). Interprocessor direct communication is prohibited.
- Records transmitted at each input phase (including the initial input file) have already sorted in the block.

Note that the restrictions affect the number of required data-transfers in at most a constant factor.

By the second assumption, the number of all possible initial permutations of N input records on the disk is the same as the number of all allocations of N input records to N/B initial blocks, $\frac{N!}{B!^{N/B}}$. In order to identify the total ordering,

$$\Omega\left(\left\lceil \log_2 \frac{N!}{B!^{N/B}} \right\rceil\right) = \Omega(N \log(N/B))$$

comparisons of two records are required in the worst case.

Let us suppose an optimal sorting algorithm with respect to the number of data transfers. The optimality of the algorithm is preserved under the additional assumptions:

- At each input phase, the processor into which the block is input completely identifies the relative total ordering of all records (including newly input records) by comparisons.
- At each output phase, records on the output block reside in the sorted order.

Consider an input phase of sorting by this algorithm just when a block is input from the disk into the internal memory of a processor. By the assumption, the total order of $M - B$ records which have resided on the memory since before the phase is known. The total order of B records on the block is also known. Merging two runs, one of which is length M and the other is length B, can be performed with at most

$$O\left(\log\binom{M+B}{B}\right) = O\left(B\log(M/B)\right)$$

comparisons. A sample routine for this merge procedure is:

(1) Choose every $\lceil M/B \rceil$-th element of the run of length M as a representative.

(2) Make a run of the representative elements (of length $O(B)$).

(3) Merge the run of length B and the run of representatives. (This requires $O(B)$ comparisons.)

(4) Search for the position of each element of the run of length B by binary search in the corresponding sub-run (of length $\lceil M/B \rceil$), each of whose terminal elements is a representative. (This requires $O(\log(M/B))$ comparisons for each element).

Since $O(B\log(M/B))$ comparisons are enough to determine the relative total order of the records on the internal memory, no more comparisons at each input phase

are meaningful. At an output phase, there comes no new comparison to be done. Thus, at least

$$\frac{\Omega(N\log(N/B))}{O(B\log(M/B))} = \Omega\left(\frac{N}{B}\frac{\log(N/B)}{\log(M/B)}\right)$$

phases of data transfer is required for sorting.

LEMMA IV.3. *The number data transfers required for sorting N records on a bus-connected two-level memory without broadcasting is* $\Omega\left(\frac{N}{B}\frac{\log(N/B)}{\log(M/B)}\right)$.

This bound is equal to the bound for uniprocessor two-level memory. Although we show the bound only for worst case, the same bound can be derived for average case with input records uniformly distributed in one of the $N!$ possible permutations.

3.3. Optimal algorithms. As shown in Lemma IV.3, the lower bound of the number of data transfer on a parallel two-level memory without broadcasting does not depend on the number of processors p. This indicates that the optimal algorithms for uniprocessor two-level memories are also optimal on parallel two-level memories without broadcasting, i.e., at most constant factor of acceleration would be possible even when one attached as many processors as he could.

In contrast, utilizing the internal memories of size M each distributed on p processors effectively, it may be possible to do sorting on a parallel two-level memory with broadcasting as if it were a multiprocessor shared two-level memory of size pM. In fact, the lower bound shown in Corollary IV.2 can be achieved by executing an algorithm of the well-known distribution sort on the model with broadcasting.

One trivial but important property of conventional uniprocessor two-level memories is that *internal sort* can be done without data transfer. Namely, sorting of M records, which just fit into the internal memory, can be done with as many data transfers as at least required for read records from the disk and write the result, $2\lceil M/B\rceil$.

We take up a corresponding sorting problem on the parallel model, *on-memory sort*. On-memory sort is sorting of pM records, which can be distributed into the internal memories of p processors. We show here that on-memory sort can be done with $O(pM/B)$ data transfers.

The problem definition of on-memory sort is stated as:

Problem Instance: The pM records reside in some $p(\frac{M}{B})$ sectors on the disk.

Goal: The records reside in some contiguous $p(\frac{M}{B})$ sectors on the disk in sorted order.

We present an on-memory enumeration sort algorithm, as an example of on-memory sort. We first argue on sorting $p(M-B)/2$ records.

Each of the p processors reads $(M/B-1)/2$ records allocated to it from the disk. It also prepares on the rest area of the internal memory a $\lceil \log_2 pM \rceil$-bit counter for each record input on the processor. By the assumption for the size of a record, the memory size required for the counters is at most $O(p(M-B)/2)$. Next each of the processors broadcasts in turn the records on it block by block to all processors (including itself). All of the processors receive broadcasted records, compare each of the received records and each of the records on its internal memory. When a internal record is smaller than a received record, the counter associated with the internal record is incremented by 1. Here we assume that a comparison of two distinct records must return either of "small" or "large", not "equal". This can be implemented by comparing, e.g., the addresses at the initial stage for two distinct records with the same key values.

After the counting procedure, each counter indicates the ranking of the associated record among all of the input records. Then the processors write back all pairs of the records and their rankings to the disk. After that, the file server broadcasts all of such pairs, and at the same time, the ith processor $(i = 0, ..., p-1)$ receives

all of the $\{\frac{1}{2}(M-B)i+1$-th to the $\frac{1}{2}(M-B)(i+1)\}$ records and keep them on the memory. Finally the processors output the sorted blocks serially (from the zeroth processor) to the disk.

This procedure performs sorting of $p(M-B)/2$ records with $O(pM/B)$ data transfers. Sorting of pM records is done by dividing the input records into four parts, sorting each of the parts by the procedure, and merging the four sorted run on one of the processors. The on-memory sorting algorithm shown here is a variant of enumeration sort [Knu73], but the discussions hereafter hold for any on-memory sorting algorithm with data-transfer cost $O(pM/B)$.

Next we show how to execute the external distribution sorting algorithm presented by Aggarwal and Vitter [AV88] on a parallel two-level memory with broadcasting, within the same data transfer cost up to a constant factor. We utilize the on-memory sorting algorithm instead of internal sort in the original algorithm. The modified external distribution algorithm is as follows.

For simplicity, we assume that M/B is a perfect square and we use S to denote the quantity $\sqrt{M/B}$. Records $b_1, \ldots, b_S$ ($b_{i_1} < b_{i_2}$ if $i_1 < i_2$) are called S *approximate partitioning elements* of the input N records if the number of records in interval $(b_{i-1}, b_i]$, say N_i, satisfies[1],

$$\frac{1}{2}\frac{N}{S} \leq N_i \leq \frac{3}{2}\frac{N}{S}$$

for all $i \in \{1, ..., S+1\}$. One step of the sorting algorithm is computing a set of S approximate partitioning elements of the input file and breaking it up into S roughly equal-sized "buckets". The partitioning step is repeated recursively until the size of each bucket is less than pM, and then do on-memory sort.

S approximate partitioning elements of N records can be gotten with $O(N/B)$ data transfers, as we will show later. Thus the external distribution sort is per-

[1]For completeness, we define the dummy partitioning elements $b_0 = -\infty$ and $b_{S+1} = +\infty$.

formed with

$$O\left(\frac{N}{B}\frac{\log(N/B)}{\log(pM/B)}\right)$$

data transfers.

Now we describe how to get S approximate partitioning elements. Note that the kth element of n records can be computed with $O(n/B)$ data transfers for any $k(< n)$ [BFP+73]. We first sort every pM records and pick up every $S/4$-th element of them. This can be done "on memory" with $O(N/B)$ data transfers. Next we choose the Ni/S^2-th element of the $4N/S$ records, and set it as b_i, for each $i = 1, \ldots, S$. This takes $O(S \cdot (N/S)/B) = O(N/B)$ data transfers, and the b_i's satisfy the requirement.

THOEREM IV.4. *The average-case and worst-case number of data transfers required for sorting N records on a bus-connected parallel two-level memory is*

$$\Theta\left(\frac{N}{B}\frac{\log(N/B)}{\log(pM/B)}\right) \quad \text{(on models with broadcasting)},$$
$$\Theta\left(\frac{N}{B}\frac{\log(N/B)}{\log(M/B)}\right) \quad \text{(on models without broadcasting)}.$$

Distribution sort is accelerated by "multiprocessing" on parallel models with broadcasting. Memories distributed to processors are utilized as if they were "shared" by the processors. However, it essentially depends on the original algorithm whether such acceleration is possible or not. In fact, it is open if we can achieve the same acceleration by multiprocessing with the merge sort, another well-known external sorting algorithm, since it is open if on-memory merge procedure can be done with $O(pM/B)$ data transfers.

4. Permuting

4.1. Problem definition. Permuting is sorting of N records whose keys are $\{1, 2, \ldots, N\}$. It can be regard as sorting where the order of each record is known

and the goal position of each record on the disk can be determined without any comparison.

4.2. The lower bound. The following result on the data transfer complexity of permuting N records on (uniprocessor) two-level memories has been shown by Aggarwal and Vitter [AV88].

PROPOSITION IV.5 (AGGARWAL, VITTER 1988). *The average-case and worst-case number of I/Os (data transfers) required for permuting N records is*

$$\Theta\left(\min\{N, \frac{N}{B}\frac{\log(N/B)}{\log(M/B)}\}\right).$$

The next lower bound follows from this immediately.

COROLLARY IV.6. *The number of data transfers required for permuting N records on a bus connected two-level memory is*

$$\Omega\left(\min\{N, \frac{N\log(N/B)}{B\log(pM/B)}\}\right).$$

Again, an improved bound can be derived on models without broadcasting. For simplicity, we pose a restriction to the model.

- Data transfer is done as either *input* (from the disk of the file server to the internal memory of a processor) or *output* (from the internal memory to the disk).

The restriction deteriorate the number of data transfers required for permuting at most a constant factor. Additionally. we may assume that

- At each stage of permuting a record exits at just one cell of either on the disk or on the internal memory of a processor. In other words, when a record is transmitted, the original one must be removed.

This assumption does not affect the lower bound since any input record is output with no modification. For any optimal permuting algorithm, there might be an

imaginary optimal algorithm which does not hold any copy of a record during the permuting procedure.

A *permutation* of records implemented at some stage of permuting is the order of records on the internal memories and the disk, where all data other than the records are neglected.

Consider the number of permutations which can be generated with T data transfers. At the tth I/O stage which is input, there are at most $N/B + t - 1$ choices of the sector to be read, and p choices of the processor to receive the block. There are also $\binom{M}{B}$ choices how to put the input B records on the memory. If the block is firstly read from the disk, $B!$ permutations of the record contribute the number of generated permutations. Thus the number of all permutations generable by t I/Os are, at most

$$(N/B + t)B!p\binom{M}{B}$$

times of that by $t - 1$ I/Os when a block is firstly read from the disk, and

$$(N/B + t)p\binom{M}{B}$$

times of that at other input stage.

At the tth I/O stage which is output, there are $N/B+t$ choices of which sector on the disk to write the block, and p choices of the processor which writes the block. There are $\binom{M}{B}$ choices in the combination of the records written out from the internal memory of the processor. Thus the number of all permutations generable by t I/Os are, at most

$$(N/B + t)p\binom{M}{B}$$

times of that by $t - 1$ I/Os when a block is written to the disk. The number of all permutations which can be generated by T data transfers is, therefore, at most

$B!^{N/B}((N/B+t)p\binom{M}{B})^T$. With a trivial upper bound for t,

$$(N/B+t) \leq N(1+\log N),$$

a necessary condition that the number of permutations generable by T data transfers is larger than $N!$ is written as

$$B!^{N/B}\left(N(1+\log N)p\binom{M}{B}\right)^T \geq N!.$$

Applying Stirling's formula, we get.

$$T(\log N + \log p + B\log\frac{M}{B}) = \Omega(N\log\frac{N}{B}),$$

namely,

$$T = \Omega\left(\min\{N, \frac{N\log(N/B)}{B\log(M/B)}\}\right).$$

This equals the bound in Proposition IV.5, i.el, at most constant factor of acceleration is possible by multiprocessing on the models without broadcasting.

Note that the bound also holds for sorting with operations other than comparison of two records available, since permuting is a special case of sorting where the total order of all input records is known.

4.3. Optimal algorithms. Permuting is a special case of sorting, and hence, the optimal sorting algorithm shown in Section 3.3 is also an algorithm for permuting. On the other hand, there exists a trivial algorithm with $O(N)$ data transfers using only one processor. The lower bound shown in Section 4.2 is achieved by using $O(N)$ naive algorithm when $N < \frac{N\log(N/B)}{B\log(pM/B)}$ on models with broadcasting or $N < \frac{N\log(N/B)}{B\log(M/B)}$ on models without broadcasting, and using the optimal sorting algorithm otherwise.

THOEREM IV.7. *The average-case and worst-case number of data transfers required for permuting N records on a bus-connected parallel two-level memory is*

$$\Theta\left(\min\{N, \tfrac{N}{B}\tfrac{\log(N/B)}{\log(pM/B)}\}\right) \quad \text{(on models with broadcasting)},$$
$$\Theta\left(\min\{N, \tfrac{N}{B}\tfrac{\log(N/B)}{\log(M/B)}\}\right) \quad \text{(on models without broadcasting)}.$$

5. Fast Fourier transform

5.1. Problem definition. Assume that N is a power of 2, and let n denote $\log_2 N$. The 2^n point FFT (fast Fourier transform) digraph is defined as:

Problem Instance: N records $X_{k,0}$ $(0 \le k \le N-1)$ reside on the contiguous N/B sectors at the beginning of the disk of the file server.

Internal Operations: $X_{k,l}$ and $X_{k\oplus 2^{l-1},l}$ are computed from $X_{k,l-1}$ and $X_{k\oplus 2^{l-1},l-1}$ $(1 \le l \le \log_2 N)$. Here $\oplus$ is the binary operator which represents the bitwise exclusive-or of the standard binary representations of two nonnegative integers.

Goal: N records $X_{k,\log_2 N}$ $(0 \le k \le N-1)$ reside on the contiguous N/B sectors at the beginning of the disk of the file server.

In the discussions of the lower bound of the number of data transfers for FFT hereafter, we consider only *exchange switching*. Namely either of

$$\begin{cases} X_{k,l} & \leftarrow X_{k,l-1} \\ X_{k\oplus 2^{l-1},l} & \leftarrow X_{k\oplus 2^{l-1},k-1} \end{cases} \quad \text{(direct)}$$

or

$$\begin{cases} X_{k,l} & \leftarrow X_{k\oplus 2^{l-1},l-1} \\ X_{k\oplus 2^{l-1},l} & \leftarrow X_{k,l-1} \end{cases} \quad \text{(exchange)}$$

can be performed for two records $X_{k,l-1}$ and $X_{k\oplus 2^{l-1},l-1}$ on the internal memory of a processor. Note that a data-transfer algorithm must decides which sector on the disk a block is read or written and which processor receives each record of

the block being transmitted on the bus independent of the configuration (whether direct or exchange) of each exchange switch.

Adopting the notation of permutation algebra used in the theory of switching networks [HJ88b], the 2^n-point FFT digraph is represented as

$$\mathrm{FFT}_{(n)} = E_{(1)}E_{(2)}\dots E_{(n)}.$$

$E_{(l)}$ is called the lth order exchange switch, and defined as

$$E_{(l)} = \{\epsilon_{(l)}, \iota\} = \{\sigma^{-l}\epsilon_{(1)}\sigma^{l}, \iota\},$$

where ι is the identity permutation, σ is the perfect shuffle permutation, and $\epsilon_{(l)}$ is the lth exchange permutation. ι, σ and $\epsilon_{(l)}$ are define as

$$\begin{aligned}\sigma(k) &= \{b_{n-1}, \cdots, b_1, b_n\}\\ \epsilon_{(l)}(k) &= k \oplus 2^{l-1}\\ &= \{b_n, \dots, b_{l+1}, \bar{b}_l, b_{l-1}, \dots, b_1\}\end{aligned}$$

using the binary representation of k,

$$k = \{b_n, b_{n-1}, \dots, b_1\} = b_n 2^{n-1} + b_{n-1}2^{n-2} + \cdots + b_1$$

($\bar{b}_l$ is the complement of b_l).

5.2. The lower bound. The 2^n-point FFT digraph is equivalent to the indirect binary n-cube network $C_{(n)}$, or the inverse of the Omega network Ω^{-1} [Tom86b]. We can construct a full connection network by stacking three FFT digraphs in cascade. Here we call a multistage network composed of exchange switches (2×2 crossbar switches) with N inputs and N outputs *full connection network* if the network can compute any of $N!$ permutations from the inputs to the outputs by setting the configurations of the exchange switches. Thus a lower bound for an arbitrary full connection network also holds for FFT digraphs.

First we show a lower bound for full connection networks on models without broadcasting. Just like in Section 4.2, we pose two restrictions to the model.

- Data transfer is done as either input or output. Direct communication between two processors is prohibited.
- A record exits at just one cell of either on the disk or on the internal memory of a processor.

Consider a full connection network, and an optimal data-transfer algorithm which realizes the network. The algorithm determines which sector on the disk to be transferred, which processor to receive or send the block, and at which cell on the disk the record received by the file server is written, independent of the contents of the input records. Hence the factor of permutations generable by an input phase is, $B!\binom{M}{B}$ when the record is firstly read, and $\binom{M}{B}$ otherwise, and the factor by an output phase is $\binom{M}{B}$. Thus the number of permutations generable by T I/Os is at most $B!^{N/B}\binom{M}{B}^T$. The necessary condition that it is larger than $N!$ leads

$$T = \Omega\left(\frac{N}{B}\frac{\log(N/B)}{\log(M/B)}\right).$$

Next we consider on models with broadcasting. At a data-transfer phase which is input, B records on the block are transmitted to the bus by the file server. Let $B_i(\geq 0)$ be the number of records to be received by the ith processor $(i = 0, \ldots, p-1)$. By the assumption that there is no copy of records. the sets of records received by distinct two processors each are disjoint, and therefore $B_0 + B_1 + \cdots + B_{p-1} = B$ holds. Note that $B_0, B_1, \ldots, B_{p-1}$ are non-adaptively fixed. The number of choices how to send records are

$$\prod_{i=0,\ldots,p-1}\binom{M}{B_i},$$

and this has the maximum value,

$$\binom{M}{1}^B = M^B$$

when $p \geq B$, or

$$\binom{M}{\lfloor B/p \rfloor + 1}^{B \bmod p} \binom{M}{\lfloor B/p \rfloor}^{p-B \bmod p} < \binom{M}{\lfloor B/p \rfloor + 1}^p$$

when $p < B$. The number of generable permutation becomes

$$\min\left\{\binom{M}{\lfloor B/p \rfloor + 1}^p, M^B\right\}$$

times at an input phase of a record and $B!$ is multiplied up when the record is firstly read. Thus the number of permutations generable by T I/Os is at most

$$B!^{N/B} \min\left\{\binom{M}{\lfloor B/p \rfloor + 1}^p, M^B\right\}^T.$$

The necessary condition that this becomes larger than $N!$ leads

$$T = \Omega\left(\frac{N}{B}\frac{\log(N/B)}{\log \min\{M, pM/B\}}\right) = \Omega\left(\frac{N}{B}\frac{\log(N/B)}{\log(pM/(B+p))}\right).$$

LEMMA IV.8. *The number data transfers required for FFT digraph of N records on a bus-connected two-level memory is*

$$\Omega\left(\frac{N}{B}\frac{\log(N/B)}{\log \frac{pM}{B+p}}\right) \quad \text{(on models with broadcasting)},$$
$$\Omega\left(\frac{N}{B}\frac{\log(N/B)}{\log(M/B)}\right) \quad \text{(on models without broadcasting)}.$$

5.3. Optimal algorithms.

5.3.1. *Decomposing FFT digraphs.* Let m be a divisor of n. 2^n-point FFT digraph can be decomposed into m iterations of $E_{(1)}E_{(2)}\cdots E_{(m)}$ and the mth power

of the inversed perfect shuffle permutation, σ^{-m}.

$$\begin{aligned}
\mathrm{FFT}_{(n)} &= E_{(1)}E_{(2)}\cdots E_{(m)}E_{(m+1)}\cdots E_{(n)} \\
&= E_{(1)}\cdots E_{(m)}\sigma^{-m}\{E_{((m+1)-m)}\cdots E_{(n-m)}\}\sigma^{m} \\
&= E_{(1)}\cdots E_{(m)}\sigma^{-m}\{E_{(1)}\cdots\sigma^{-m}\{E_{(1)}\cdots E_{(m)}\}\sigma^{m}\cdots\}\sigma^{m} \\
&= (E_{(1)}\cdots E_{(m)})\sigma^{-m}\cdots\sigma^{-m}(E_{(1)}\cdots E_{(m)})\sigma^{(n/m-1)m} \\
&= (E_{(1)}\cdots E_{(m)}\sigma^{-m})^{n/m}
\end{aligned}$$

The permutation $\left(E_{(1)}\cdots E_{(m)}\right)(k)$ affects only the least significant m bits of the binary representation of k, and can be regard that it performs $\mathrm{FFT}_{(m)}$ for each input records whose most significant $n-m$ bits are the same independently. Namely a 2^n-point FFT digraph can be decomposed into n/m stages as

$$\mathrm{FFT}_{(n)} = (\mathrm{FFT}_{(m)}\sigma^{-m})^{n/m}$$

where each stage is composed of 2^{n-m}-parallel 2^m-point FFT digraphs (2^n inputs total) and m inversed perfect shuffles (Figure IV.11).

5.3.2. *On-memory FFT.* We first present an algorithm for computing $\mathrm{FFT}_{(\log_2 pM)}$ of pM inputs on p-parallel two-level memory model with broadcasting. For simplicity, we may assume that p, B and M are some powers of 2 and $\log_2 M$ exactly divides $\log_2 pM$. According the decomposition in Section 5.3.1, a pM-point FFT digraph $\mathrm{FFT}_{(\log_2 pM)}$ is decomposed into

$$\mathrm{FFT}_{(\log_2 pM)} = (\mathrm{FFT}_{(\log_2 M)}\sigma^{-\log_2 M})^{\frac{\log_2 pM}{\log_2 M}}.$$

Each $\mathrm{FFT}_{(\log_2 M)}$ can be computed on the internal memory of each processors, and $\sigma^{-\log_2 M}$ can be implemented by pM/B phases of broadcasting. Thus $\mathrm{FFT}_{(\log_2 pM)}$ can be performed with

$$\frac{pM}{B}\frac{\log_2 pM}{\log_2 M}$$

data transfers.

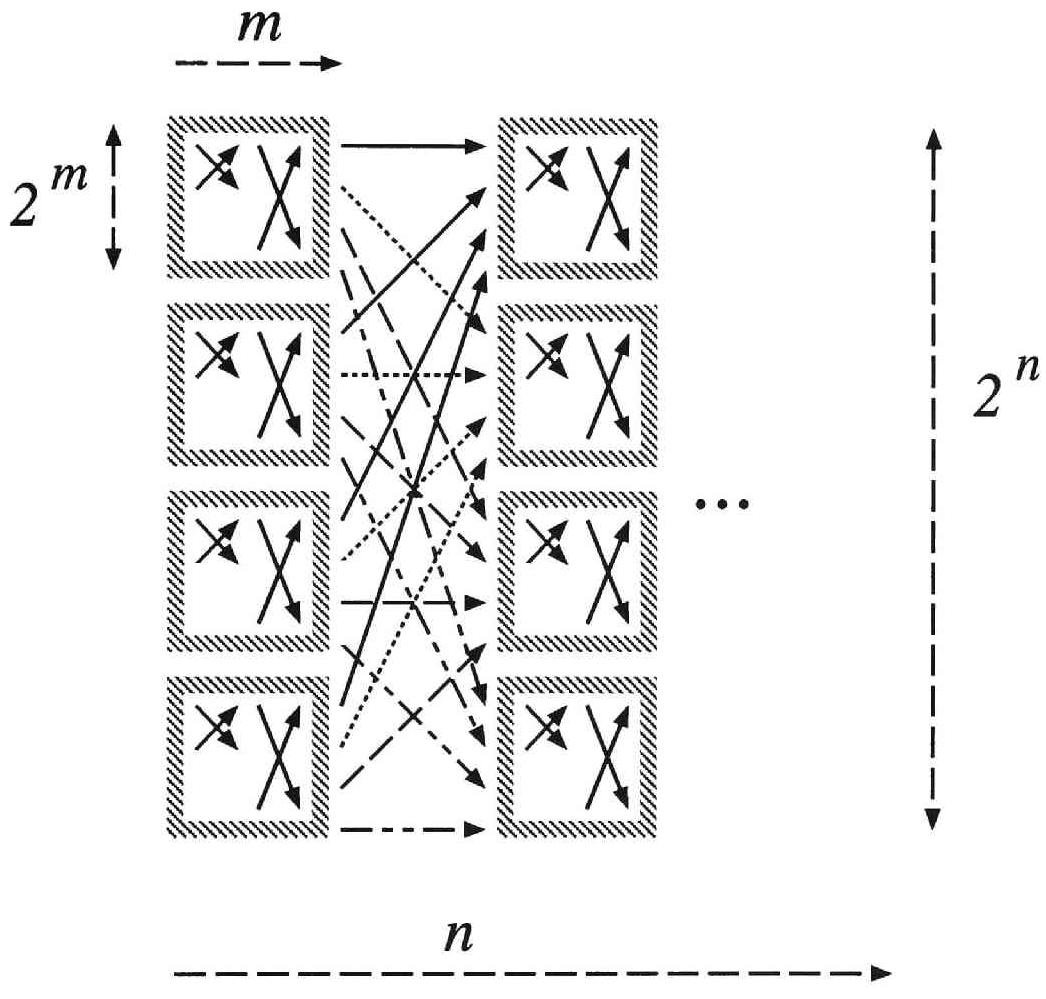


FIGURE IV.11. Decomposing an FFT digraph into sub-FFTs and shuffles.

This algorithm is an FFT algorithm for records just fit into the internal memories distributedly, and we call this *on-memory FFT*. While on-memory sorting requires $O(\frac{pM}{B})$ data transfers, which is just the same order as what is required for just load pM records into memories, it takes $\Theta(\frac{pM}{B}\frac{\log pM}{\log M})$ data transfers to perform on-memory FFT on this algorithm (The optimality of this algorithm will be shown in Section 5.3.4).

5.3.3. *Inversed shuffles.* We denote that $m = \log_2 pM$ and $h = \log_2 B$. The $(m-h)$-th power of the inversed perfect shuffle $\sigma^{-(m-h)}$ of $N(= 2^n)$ inputs is described formally as follows. We show it only for $h \geq m-h$, but it is similarly for $h < m-h$.

Problem Instance: N records resides on some N/B contiguous sectors on the disk, where a record numbered

$$\{b_n, \ldots, b_{h+1}, \underbrace{b_h, \ldots, b_1}\}$$

is at the $\{b_h, \ldots, b_1\}$-th cell of the $\{b_n, \ldots, b_{h+1}\}$-th sector.

Goal: N records resides on some N/B contiguous sectors (not necessary the same as the input sectors) on the disk, where a record numbered $\{b_n, \ldots, b_1\}$ is at the $\{b_m, \ldots, b_{m-h+1}\}$-th cell of the $\{\underline{b_{m-h}, \ldots, b_1}, b_n, \ldots, b_{m+1}\}$-th sector.

$\sigma^{-\log_2 pM}$ moves the

$$\{b_n, \ldots, b_{m+1}, b_m, \ldots, b_{h+1}, \underbrace{b_h, \ldots, b_{m-h+1}, \underline{b_{m-h}, \ldots, b_1}}\}\text{-th}$$

record of inputs to the

$$\{\underline{b_{m-h}, \ldots, b_1}, b_n, \ldots, b_{m+1}, \underbrace{b_m, \ldots, b_{h+1}, b_h, \ldots, b_{m-h+1}}\}\text{-th}$$

position of the outputs (Underbraces denote intrablock addresses).

Let us show an algorithm for $\sigma^{-(m-h)} = \sigma^{-\log_2 pM/B}$ of N input records.

Partition the internal memories of the processors (size pM) into memory blocks of size B each, and number them 0 to $pM/B - 1$, as the $i$the processor ($i = 0, \ldots, p-1$) has memory blocks numbered

The algorithm repeats the following procedure for all ($2^{n-m} = N/pM$) combinations of indices b_i such that

$$\begin{aligned} \{b_n, \ldots, b_{m+1}\} &= \{0, \ldots, 0\} \ , \ldots , \ \{1, \ldots, 1\} \\ &= \quad 0 \quad\quad , \ldots , \ 2^{n-m} - 1. \end{aligned}$$

(1) $2^{m-h} = pM/B$ input records numbered $\{b_n, \ldots, b_{m+1}, \underline{x_{m-h}, \ldots, x_1}\}$ are broadcasted on the bus, where

$$\begin{aligned} \{\underline{x_{m-h}, \ldots, x_1}\} &= \{0, \ldots, 0\} \ , \ldots , \ \{1, \ldots, 1\} \\ &= \quad 0 \quad\quad , \ldots , \ 2^{m-h} - 1. \end{aligned}$$

(2) Each processor put the $\{d_{h-(m-h)}, \ldots, d_1, c_{m-h}, \ldots, c_1\}$-th record of a input block numbered $\{b_n, \ldots, b_{m+1}, \underline{x_{m-h}, \ldots, x_1}\}$ is put on the

$$\{\underline{x_{m-h}, \ldots, x_1}, d_{h-(m-h)}, \ldots, d_1\}\text{-th}$$

cell of the memory block numbered $\{c_{m-h}, \ldots, c_1\}$.

(3) Records on each memory block are written out into the disk. The memory block numbered $\{c_{m-h}, \ldots, c_1\}$ is moved to the $\{c_{m-h}, \ldots, c_1, b_n, \ldots, b_{m+1}\}$-th sector of the output area on the disk.

All blocks on the disk are read just once, and as many blocks are written back to the disk. The number of required data transfers is $2N/B$.

5.3.4. *An optimal FFT algorithm.* N-point FFT digraph is decomposed into $\log_2 N / \log_2 pM$ iterations of N/pM-parallel pM-point FFT digraphs $\mathrm{FFT}_{(\log_2 pM)}$ and the $\log_2 pM$-th power of the inversed perfect shuffle $\sigma^{-\log_2 pM}$,

$$\mathrm{FFT}_{(\log_2 N)} = \left(\mathrm{FFT}_{(\log_2 pM)}\ \sigma^{-\log_2 pM}\right)^{\frac{\log_2 N}{\log_2 pM}}.$$

N/pM-parallel $\mathrm{FFT}_{(\log_2 pM)}$ requires $\frac{N}{pM}\frac{pM}{B}\frac{\log_2 pM}{\log_2 M} = O\left(\frac{N}{B}\frac{\log pM}{\log M}\right)$ data transfers, and $\frac{\log_2 pM}{\log_2 pM/B}$ iterations of N-input $\sigma^{-\log_2 pM/B}$ requires $O(\frac{N}{B}\frac{\log pM}{\log pM/B})$ data transfers. Thus $\mathrm{FFT}_{(\log_2 N)}$ can be performed with

$$O\left(\frac{N}{B}\frac{\log N}{\log pM}\left(\frac{\log pM}{\log M}+\frac{\log pM}{\log \frac{pM}{B}}\right)\right) = O\left(\frac{N}{B}\left(\frac{\log \frac{N}{B}}{\min\{M, \frac{pM}{B}\}}\right)\right)$$

data transfers. With some algebraic manupulations, this is equal to the lower bound shown in Lemma IV.8. Namely this algorithm is optimal on models with broadcasting.

On the other hand, the lower bound on models without broadcasting in Lemma IV.8 does not depend on the number of processors p, and hence the above algorithm with $p = 1$ works as an optimal algorithm for models without broadcasting. We now get:

THOEREM IV.9. *The number of data transfers required for computing an N-point FFT digraph is*

$$\Theta\left(\frac{N}{B}\frac{\log(N/B)}{\log \frac{pM}{B+p}}\right) \quad \text{(on models with broadcasting)},$$
$$\Theta\left(\frac{N}{B}\frac{\log(N/B)}{\log(M/B)}\right) \quad \text{(on models without broadcasting)}.$$

6. Matrix transposition

6.1. Problem definition. Assume that N is a power of 2. The problem definition of transposing an $N_0 \times N_1$ matrix is stated as:

Problem Instance: An $N_1 \times N_0$ matrix $A = (A_{i,j})$ of $N = N_0N_1$ records are stored in row-major order on the contiguous N/B sectors at the beginning of the disk of the file server. The internal memory is empty.

Internal Operations: Each processor moves of a record on the internal memory into some cell of the memory.

Goal: The internal memory of each processor is empty, and the transposed matrix $A^T = (A_{j,i})$ resides on the contiguous N/B sectors at the beginning of the disk.

6.2. The lower bound. Noting that matrix transposition is a special case of permuting, the bound shown in Section 4.2 holds for matrix transposition. Furthermore, we can get more strict lower bound using a potential function argument introduced by Floyd [Flo72]. Without loss of generality, we may pose the restrictions used in Section 4.2. We show first a lower bound on models without broadcasting.

Let us provide an algorithm for matrix transposition. We define the ith *target group*, for $1 \leq i \leq N/B$, to be the set of records that will be in the ith sector at the goal stage. Let f be a continuous function defined as

$$f(x) = \begin{cases} x \log x, & \text{if } x > 0; \\ 0, & \text{otherwise.} \end{cases}$$

The *togetherness rating* of the kth sector on file server at time t, $C_k(t)$, is defined as

$$C_k(t) = \sum_{1 \leq i \leq N/B} f(x_{i,k}),$$

where $x_{i,k}$ is the number of records belonging to the ith target group on the kth sector after t data transfers. Similarly the togetherness rating of the internal memory of the lth processor at time t, $\hat{C}_l$, is defined as

$$\hat{C}_l(t) = \sum_{1 \leq i \leq N/B} f(y_{i,l}),$$

where $y_{i,l}$ is the number of records belonging to the ith target group on the internal memory of lth processor after t data transfers. We define the potential at time t

to be the sum of the togetherness ratings

$$\mathrm{POT}(t) = \sum_{0 \le l \le p-1} \hat{C}_l(t) + \sum_{k \ge 1} C_k(t).$$

We denote by T the total number of I/Os performed by the end of the algorithm. At time T each of the first N/B sectors has togetherness rating $C_k(T) = B \log B$, and other sectors and the internal memory are empty; namely,

$$\mathrm{POT}(T) = N \log B.$$

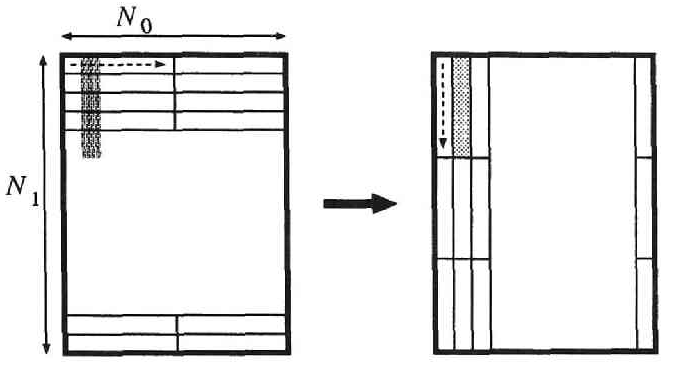
Next let us compute the initial potential $\mathrm{POT}(0)$. If $B < \min\{N_0, N_1\}$, no target group has two records that are initially in the same sector (Figure IV.12 (a)). If $\min\{N_0, N_1\} \le B \le \max\{N_0, N_1\}$, each target group is partitioned into $\min\{N_0, N_1\}$ equal-sized groups, such that the records in the same group initially reside in the same block (Figure IV.12 (b)). If $B > \max\{N_0, N_1\}$, there are N/B groups (Figure IV.12 (c)). This gives

$$\mathrm{POT}(0) = \begin{cases} 0, & \text{if } B < \min\{N_0, N_1\}; \\ N \log(B/\min\{N_0, N_1\}), & \text{if } \min\{N_0, N_1\} \le B \le \max\{N_0, N_1\}; \\ N \log(B^2/N), & \text{if } \max\{N_0, N_1\} < B. \end{cases}$$

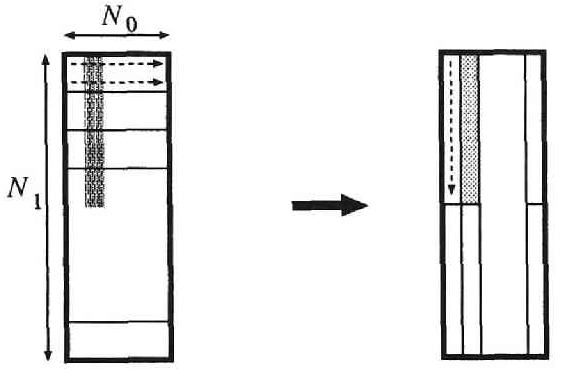
When a block is output from internal memory of a processor to disk at time t, the potential function does not increase. Let us assume that the ith data transfer is an input from the kth sector of the disk to the internal memory of the lth processor. After the input the togetherness rating $C_k(t)$ is 0. The increase in potential $\nabla\mathrm{POT}(t)$ is thus

$$\nabla\mathrm{POT}(t) = \hat{C}_l(t) - \hat{C}_l(t-1) - C_k(t-1).$$

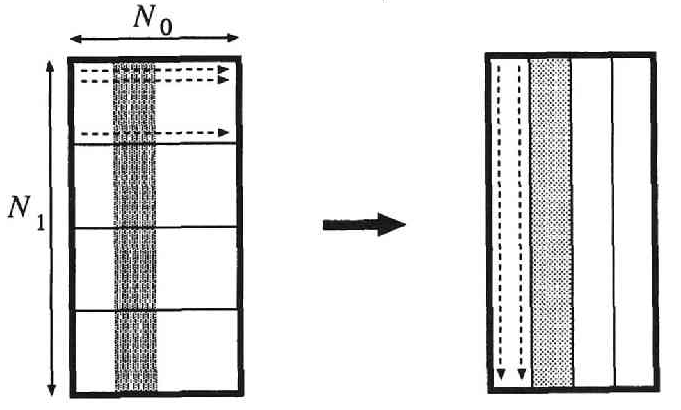
The contribution of a target group to the togetherness when records belonging to the same target group are "merged". We denote the number of records belonging to the ith target group on the internal memory of the l the processor as $\hat{y}_i$, and



(a) $B < \min\{N_0, N_1\}$



(b) $N_0 \leq B \leq N_1$



(c) $\max\{N_0, N_1\} < B$

FIGURE IV.12. Matrix transposition and the target groups.

the number of records belonging to the ith target group on the kth sector at time t as $\tilde{y}_i$. Then

$$\nabla\mathrm{POT}(t) = \sum_{1\le i\le N/B} \{f(\hat{y}_i + \tilde{y}_i) - f(\hat{y}_i) + f(\tilde{y}_i)\},$$

where

$$\sum_{1\le i\le N/B} \hat{y}_i \le M - B \quad \text{and} \quad \sum_{1\le i\le N/B} \tilde{y}_i \le B.$$

A simple convexity argument shows that $\nabla\mathrm{POT}(t)$ is maximized when $\hat{y}_i = \frac{(M-B)B}{N}$ and $\tilde{y}_i = B^2/N$, for each $1 \le i \le N/B$. Using the inequality

$$\begin{aligned} f(x+y) - f(x) - f(y) &= x\log(1+\frac{y}{x}) + y\log(1+\frac{x}{y}) \\ &\le \frac{y}{\ln 2} + y\log(1+\frac{x}{y}) \\ &= O\left(y\log(1+x/y)\right), \end{aligned}$$

we get

$$\nabla\mathrm{POT}(t) = O(B\log(M/B)),$$

and hence

$$T = \frac{N}{B} + \Omega\left(\frac{\mathrm{POT}(T) - \mathrm{POT}(0)}{B\log(M/B)}\right).$$

LEMMA IV.10. *The number data transfers required for transposing an $N_0 \times N_1$ matrix on a bus-connected two-level memory without broadcasting is*

$$\Omega\left(\frac{N}{B}\left(1 + \frac{\log\min\{M, N_0, N_1, N/B\}}{\log(M/B)}\right)\right).$$

Note that a lower bound for p-processor models with broadcasting is derived by substituting pM for each M of the bound in Lemma IV.10.

6.3. Optimal algorithms. First we show an optimal algorithm satisfying the bound in the previous section for p-processor models with broadcasting.

Consider a natural numbering

$$(i-1)N_0 + j - 1 = \{i_{r_1}, \ldots, i_1, j_{r_0}, \ldots, j_1\}$$

for the elements $(A_{i,j})$ of the matrix to be transposed, where $r_0 = \log_2 N_0$, $r_1 = \log_2 N_1$, and

$$i = \{i_{r_1}, \ldots, i_1\} = \sum_\ell i_\ell 2^{\ell-1},$$
$$j = \{j_{r_0}, \ldots, j_1\} = \sum_\ell j_\ell 2^{\ell-1}$$

(braces denote binary representations). The numbering of the transposed record $(A_{j,i})$ should be

$$(j-1)N_1 + i - 1 = \{j_{r_0}, \ldots, j_1, i_{r_1}, \ldots, i_1\}.$$

Namely matrix transposition is equivalent to exchanging the upper r_0 bits and the rest r_1 bits of the binary representation of the natural numbering. Remember the three cases of the ordering of N_0, N_1 and block size B shown in Figure IV.12.

When $B < \min\{N_0, N_1\}$, transposition of the matrix can be performed by blockwise reordering, h shuffle (the hth power of the perfect shuffle), and again blockwise reordering (Figure IV.13 (a)). Using the algorithm for the $(m-h)$-th power of the perfect shuffle presented in Section 5.3.3[2], the total number of required data transfers is $O(N/B(1 + h/(m-h))) = O(\frac{N}{B}\frac{\log pM}{\log(pM/B)})$.

When $N_0 \leq B \leq N_1$, transposition can be performed by r_0 shuffle (Figure IV.13 (b)), and this needs $O(N/B(1 + r_0/(m-h))) = O(\frac{N}{B}(1 + \frac{\log N_0}{\log(pM/B)}))$. Note that the algorithm is reversible and hence applicable to the case when $N_1 \leq B \leq N_0$; this needs $O(\frac{N}{B}(1 + \frac{\log N_1}{\log(pM/B)}))$ data transfers.

[2]The algorithm is stated as inversed shuffle, but it is easily verified that the algorithm is reversible.

When $\max\{N_0, N_1\} < B$, transposition can be performed by intrablock reordering, $n-h$ shuffle, and again intrablock reordering (Figure IV.13 (c)), and this needs $O(N/B(1+(n-h)/(m-h))) = O(\frac{N}{B}(1+\frac{N/B}{\log(pM/B)}))$.

On models without broadcasting, the above algorithm for uniprocessor ($p=1$) works with the same data transfers as the lower bound shown in Lemma IV.10. We now get:

THOEREM IV.11. *The number of data transfers required for transposing an* $N_0 \times N_1$ *matrix is*

$$\Theta\left(\frac{N}{B}\left(1+\frac{\log\min\{pM,N_0,N_1,N/B\}}{\log(pM/B)}\right)\right) \quad \text{(on models with broadcasting)},$$
$$\Theta\left(\frac{N}{B}\left(1+\frac{\log\min\{M,N_0,N_1,N/B\}}{\log(M/B)}\right)\right) \quad \text{(on models without broadcasting)}.$$

7. Standard matrix multiplication

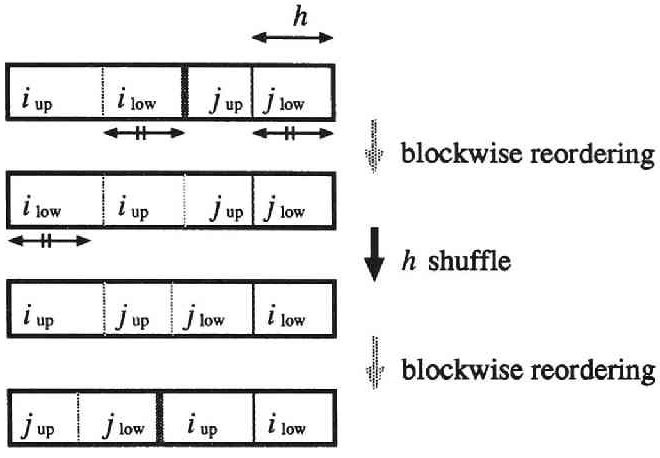
7.1. Problem definition. The problem definition of *standard matrix multiplication* is stated as:

Problem Instance: The internal memory of each processor is empty. The elements of an $N_1 \times N_1$ matrix $A = (A_{i,k})$ and an $N_1 \times N_1$ matrix $B = B_{k,j}$ are stored in the first $(2{N_1}^2)/B$ sectors of the file server in row-major order.

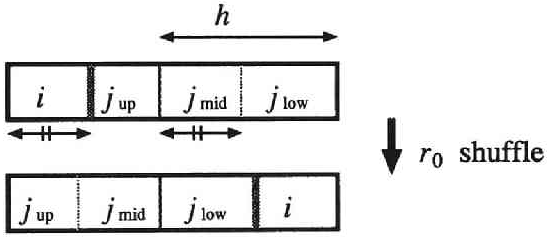
Internal Operations: Each processor reads entries of A and B, and partial sums for C. Here a partial sum of $C_{i,j}$ on $S(\subseteq \{1,\ldots,N_1\})$ is defined as

$$C_{i,j}(S) = \sum_{k\in S} A_{i,k}B_{k,j}.$$

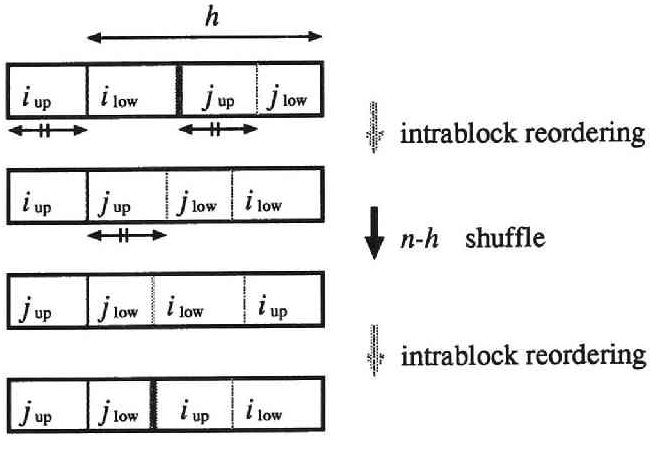
It multiplies $A_{i,k}$ and $B_{k,j}$, both of which reside on the internal memory, to get a partial product $A_{i,k}B_{k,j}$, and put it on a cell as $C_{i,j}(\{k\})$. It also adds two partial sums $C_{i,j}(S_1)$ and $C_{i,j}(S_2)$ to get $C_{i,j}(S_1 \cup S_2)$, when S_1 and S_2 are disjoint.



(a) $B < \min\{N_0, N_1\}$



(b) $N_0 \leq B \leq N_1$



(c) $\max\{N_0, N_1\} < B$

FIGURE IV.13. Matrix transposition by shuffles.

Goal: The internal memory of each processor is empty, and the product $C = (C_{i,j}(\{1, \ldots, N_1\}) = (\sum_{k \in \{1,\ldots,N_1\}} A_{i,k}B_{k,j})$, which is an $N_1 \times N_1$ matrix, resides on the first ${N_1}^2/B$ sectors at the beginning of the disk.

Standard matrix multiplication exactly needs ${N_1}^3$ multiplications and ${N_1}^2(N_1 - 1)$ additions. Here optimization utilizing the distribution low, like Strassen's algorithm [Str69], is not permissible.

7.2. The lower bound. Let us first show a lower bound of standard matrix multiplication on models without broadcasting.

During discussion of the lower bound, we take notice only on computing product terms $A_{i,k}B_{k,j}$ from two entries $A_{i,k}$ and $B_{k,j}$ and summation of them on the internal memory. Summation of such partial sums computed independently and stored to the file server is neglected. Then, we may assume that each processor reads only entries of either matrix A or matrix B. Each processor can work from start to finish without reading any output of other processors. With at most constant times loss of data transfers, we may also assume that all data transfers are done with consecutive M input phases to a processor and consecutive M output phases from the processor, and that the internal memory of the processor is cleared after the output phases.

Next we show the maximum number of terms that can be computable on the internal memory of a processor with consecutive M input phases and consecutive M output phases is $\Omega(M^{3/2})$. This bound is originally proved on uniprocessor model with $B = 1$ [HK81]. Let W_A, W_B and W_C be any set of entries in A B and C, with $|W_A \cup W_B \cup W_C| \leq M$. Partition A into two classes as follows. Class A_d consists of all rows in A, each of which has at least $\sqrt{M}$ entries in W_A, and class ${A'}_d$ consists of the rest of rows in A. Accordingly, matrix C is partitioned into two blocks, A_dB and ${A'}_dB$ (Figure IV.14 (a)). Since A_d can have at most

$\sqrt{M}$ rows, and since any row of A_dB an entry in B can appear at most once (and B has no more than M entries in W_B), the maxmum number of terms in A_dB that can be obtained by multiplying elements in W_A and W_B is at most $M\sqrt{M} = M^{3/2}$ (Figure IV.14 (b)). For terms in ${A'}_dB$, each of them can be obtained by multiplying at most $\sqrt{M}$ elements in W_A and W_B, since each row in ${A'}_d$ has at most $\sqrt{M}$ elements in W_A (Figure IV.14 (b)). Therefore, in W_C, a subset of ${A'}_dB$ having no more than M elements, the maximum number of terms that can be obtained by multiplying elements in W_A and W_B is at most $M\sqrt{M} = M^{3/2}$.

In standard multiplication of two $N_1 \times N_1$ matrices, N_1^3 terms must be computed. Hence the number of required data transfers is $\Omega(\frac{{N_1}^2}{B} + \frac{M}{B}\frac{{N_1}^3}{M^{3/2}}) = \Omega(\frac{{N_1}^3}{\min\{N_1, \sqrt{M}\}B})$.

LEMMA IV.12. *The number data transfers required for standard multiplication of two $N_1 \times N_1$ matrices on a bus-connected two-level memory without broadcasting is*

$$\Omega\left(\frac{{N_1}^3}{\min\{N_1, \sqrt{M}\}B}\right).$$

Note that a lower bound for p-processor models with broadcasting is derived by substituting pM for each M of the bound in Lemma IV.12.

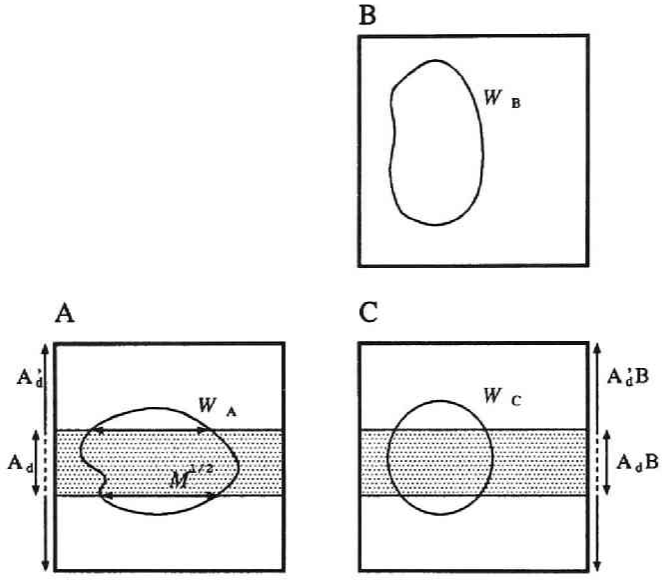
7.3. Optimal algorithms. Now we show an algorithm for multiplication of two matrices on models with broadcasting.

Repeat recursively the following partitioning until k, the size of each submatrix, is less than $\sqrt{pM}/2$ [VS90].

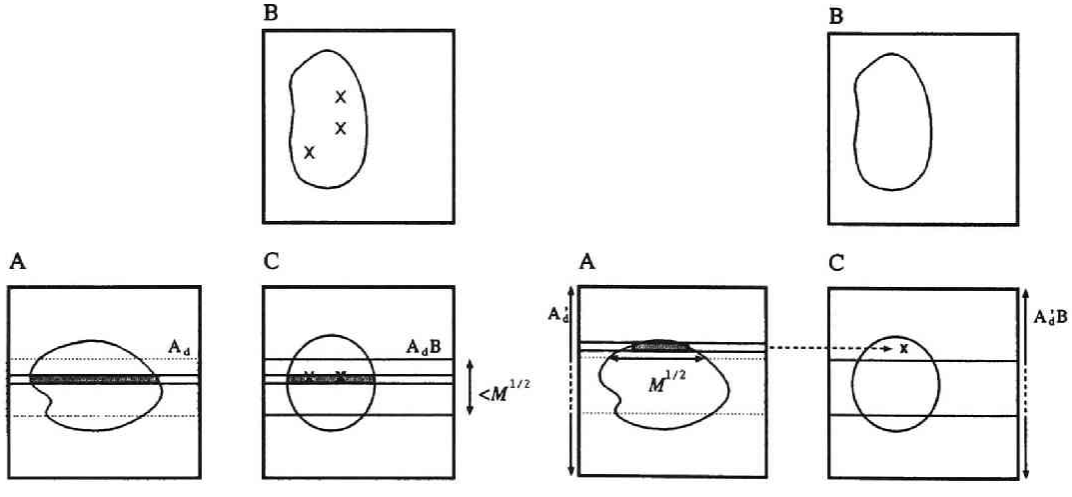
(1) Divide A and B as

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, \quad B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix},$$

and store $A_{00}, \ldots, A_{11}$, and $B_{00}, \ldots, B_{11}$ in row-major order.



(a) Partition of A.



(b) Computing A_dB (c) Computing A'_dB

FIGURE IV.14. Computing partial sums of terms on internal memory.

(2) Compute each of the products of submatrices $A_{00}B_{00}$, $A_{01}B_{10}$, ... ,$A_{10}B_{01}$, $A_{11}B_{11}$ by invoke this procedure recursively.

(3) Add the four pairs of submatrices,

$$C_{00} = A_{00}B_{00} + A_{01}B_{10}, \quad C_{01} = A_{00}B_{01} + A_{01}B_{11},$$
$$C_{10} = A_{10}B_{00} + A_{11}B_{10}, \quad C_{11} = A_{10}B_{01} + A_{11}B_{11}.$$

(4) Reposition $C_{00}, \ldots, C_{11}$ so that

$$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}$$

is stored in raw-major order.

We define $T(k)$ to be the number of data transfers used to multiply two $k \times k$ matrices. Since the number of data transfers required in the first, third and fourth steps in the above procedure is $O(k^2/B)$,

$$T(k) = 8T(k/2) + O(k^2/B)$$

holds.

Next we show how to compute

$$c_{I,J} = \sum_{K=1}^{k} a_{I,K} b_{K,J}$$

when $k \leq ..\sqrt{pM}/2$ (Figure IV.15).

(1) The ith processor ($0 \leq i \leq p-1$) reads records

$$a_{I,*}, \quad I = \frac{k}{p}i + 1, \ldots, \frac{k}{p}i + k/p$$

from the file server ($*$ denotes that the index matches any possible value).

(2) The file server broadcasts records $b_{K,*}, K = 1, \ldots, k$, sequentially. The ith processor receives them and computes

$$c_{*,I} := c_{*,I} + a_{I,K} b_{K,*};$$

for $I = \frac{k}{p}i + 1, \ldots, \frac{k}{p}i + \frac{k}{p}$.

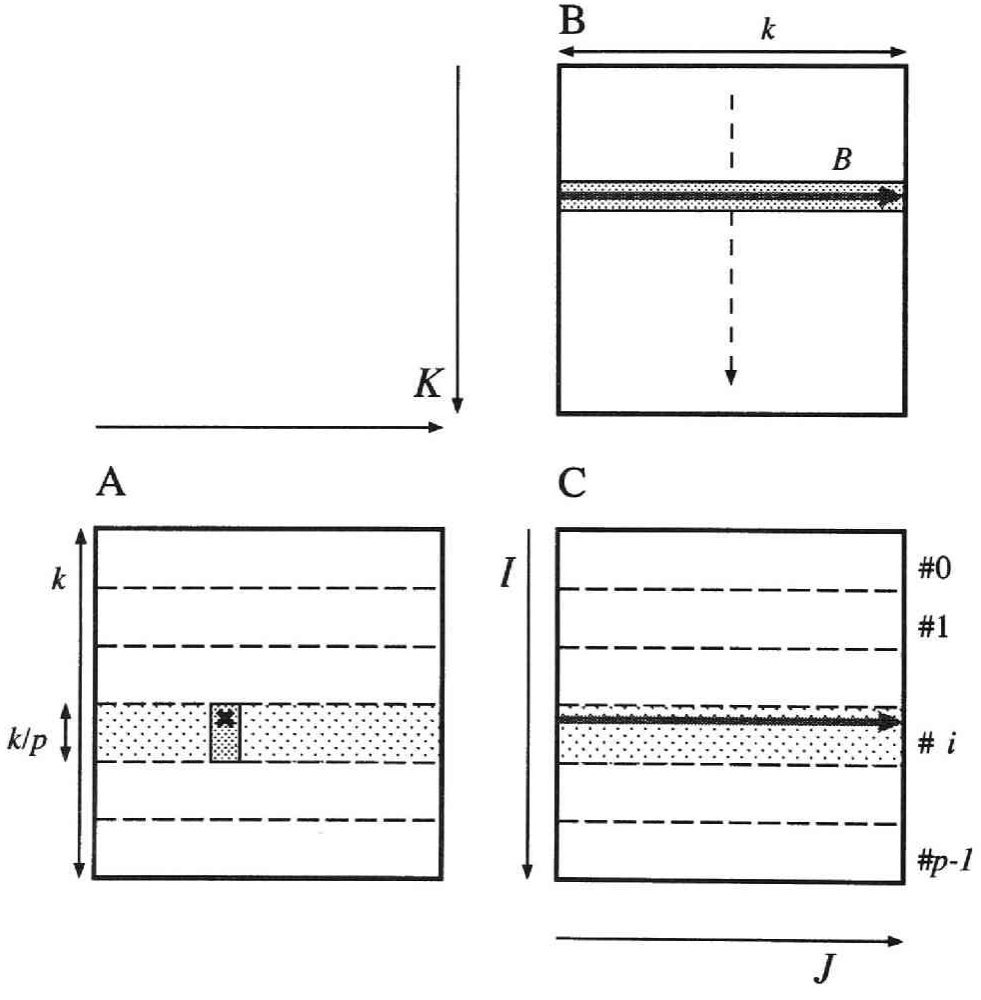


FIGURE IV.15. Computing the product of two matrices on memory.

(3) Each processor stores $c_{I,J}$ to the file server in turn.

Each steps of the procedure needs k^2/B data transfers and hence

$$T(k) = 3k^2/B = O(pM/B).$$

Thus we get

$$T(N_1) = O(\frac{{N_1}^3}{B} + \frac{{N_1}^3}{\sqrt{pM}B}).$$

On models without broadcasting, the above algorithm for uniprocessor ($p = 1$) works with the same data transfers as the lower bound shown in Lemma IV.12. We now get:

THOEREM IV.13. *The number of data transfers required for standard multiplication of two $N_1 \times N_1$ matrices is*

$$\Theta\left(\frac{{N_1}^3}{\min\{N_1, \sqrt{pM}\}B}\right) \quad \text{(on models with broadcasting,)}$$
$$\Theta\left(\frac{{N_1}^3}{\min\{N_1, \sqrt{M}\}B}\right) \quad \text{(on models without broadcasting)}.$$

Although the problem definition is concerned with only multiplication of square matrices, the results stated in this section hold for multiplication of general matrices when all of the dimensions of them are in the same order.

8. Concluding remarks

We have argued on the lower bounds of data transfer complexity and optimal algorithms which perform the bounds for sorting, permutation, FFT, etc., on bus-connected parallel two-level memories. In particular, the difference between models with broadcasting and those without broadcasting has been clearly stood out.

On a model with broadcasting, increasing the number of processors does not contribute to the power of it. Even on a model without broadcasting, the acceleration of the power is at most $O(\log p)$ for sorting and related problems. This might indicates a limit of the computation power of bus-connected parallel machines.

The acceleration for FFT is bounded by the block size B. This is because we assume that FFT is computed with the FFT digraph. The definition of the problem itself defines partly a data transfer algorithm, and we can't extract more parallelism than what is defined. In fact, for discrete Fourier transform, we can achieve acceleration of $O(\log p)$ times by utilizing an on-memory DFT algorithm with $O(pM/B)$ data transfers, although it is not optimal with respect to the number of arithmetic operations.

CHAPTER V

Use of Semiconductor Extended Storage of Vector Supercomputers as Extended Main Storage

1. Introduction

Vector supercomputers are used in various fields of science to perform large-scale numerical computations. The peak performance of floating-point operations of them now reaches more than 100 times as much as that of conventional scalar computers [Fer86, HJ88a].

Realization of not only a high performance processor but also a mass and high-speed memory system is indispensable for processing an enormous amount of numerical data. Since the memory demand for large-scale computation is far beyond the largest possible size of main memory (denoted by MM hereafter), high-grade models of vector supercomputers are equipped with semiconductor secondary memory device called extended storage (or ES for short)[ONK86, KaHMK+88, WKI86]. The capacity of ES is typically an order of magnitude greater than that of MM, and the speed of data transfer between ES and MM is several hundred times as fast as that between ES and magnetic disks.

At present, ES is generally used as a high-speed equivalent of magnetic disks. Indeed, the only way for users to access ES is to open files on it and read/write data as records via FORTRAN standard input/output statements. A programmer

cannot enjoy the vastness of ES without rewriting his program to use explicit I/Os; he has to determine the control of data transfer between ES and MM, and has to insert READ/WRITE statements appropriately into the program. This toil is too much for an ordinary user. Such rewrites may also leads bugs.

While virtual memory system, which conceals the presence of memory hierarchy from users, is commonly used on most of general-purpose scalar computers, almost all vector supercomputers works on real memory. The main reason of this is that the dynamic address transformation mechanism might significantly decrease the speed of MM access of supercomputers. It is also reported that virtual memory systems based on demand paging do not necessarily work well for large-scale numerical applications which have no locality of references [ASP82]. Our experimental performance evaluation on virtual memory system of IBM ES/3090 with vector facility (a rather slow vector computer) using expanded storage [OT90b, OT90a] also indicates that modification of algorithms, e.g. loop tiling, are indispensable to achieve enough of the performance.

In this chapter, we propose a new method, what we call "virtualization of ES" [Tsu88a], for use of ES as extended main storage. Our target is a FORTRAN program written without considering the presence of memory hierarchy, whose array data may become larger than the size of MM. The program is automatically transformed by a language processing system (e.g. a vectorizing compiler) so that it does appropriate I/O between MM and ES. Huge size of arrays that overflow the MM are detected at the transformation, and allocated as files on ES. Efficient partitioning of each array into pages is determined based on the access patterns. Codes for data transfers between MM and ES are inserted at optimal positions, considering the structure of DO loops of the program. The transformation is done without changing the order of arithmetic operations in the original program. As far as array data is concerned, users can write programs as if MM were as large

as ES. We may say that ES is virtualized as extension of MM.

Partitioning of an array affects the total performance essentially. Data of a multidimensional array divided in a standard striped partitioning over only one dimension can be accessed along not-partitioned dimensions in high-speed vector mode, but access along the partitioned dimension is done with short vector and causes transfers of data which never used. Instead we utilize tiling partitioning [Wol89]. Access of an array along any dimension invokes transfers of as much data as what are used. Vector-mode execution in the original program is preserved with the same vector length by utilizing indirect vector addressing.

We have developed a preprocessor which performs the proposed program conversion at FORTRAN source-code level. The performance of our proposed method is evaluated by converting some typical numerical programs and executing them on Hitachi's S–3800/480. The transformed code of an LU factorization program achieves $1 Gflops$, which is about 50% of the execution speed when the original one is executed normally on main memory.

In Section 2, we will describe hardware specifications and software supports of extended storages. In Section 3, we will propose our method of virtualization of ES, and compare it with conventional usages of ES. Allocation of huge arrays on ES, and management of data transfer between ES and MM will be discussed in Section 4 and in Section 5 respectively. We will present an implementation of the proposed method as a preprocessor at FORTRAN source-code level in Section 6, and will show some results of performance evaluation in Section 7.

2. Extended storage

Most vector supercomputers are equipped with high-speed semiconductor secondary storage composed of dynamic RAM chips. It is called extended storage by Hitachi, system storage by Fujitsu, extended memory unit by NEC, and solid-state

disk (SSD) by Cray. We call it simply as *extended storage (ES)* hereafter. The specifications of ES of these machines are shown in Table V.1. The capacity of ES is one digit larger than that of MM, and the speed of data transfer between the ES and the MM is several hundred times as fast as that of magnetic disks. For example, the maximum capacity of ES of Hitachi HITAC S–3800/480 is $16GB$, which is 8 times as large as the maximum capacity of MM ($2GB$). The maximum data-transfer rate on S–3800/80 is $4GB/sec.$

Ordinary users can use ES by standard FORTRAN I/O statements (READ/WRITE), just like magnetic disk. Either direct access method and sequential access method is available. Data are transferred in blocks between ES and MM. High speed in data transfers is achieved only when the size of blocks is enough large. For example, 80% of the peak performance is achieved with a block of $8MB$ on S–820/80, and a block of $32MB$ on HITAC S–3800/480.

3. Use of extended storage as extended main memory

3.1. Conventional approaches to supercomputer memory hierarchy.

Virtual memory system is commonly used as a method to execute programs which are too large to fit in the available main memory. The operating system keeps some parts of a program currently in use on MM, and the rest on the disk. The virtual address space is divided up into pages; access to an unmapped page causes the CPU to trap to the OS (page fault), and the OS fetches the referenced page from the disk. Data of a program, i.e., program text, heap data, and stack, are not distinguished by virtual memory system. Page fault may occurs at any point of execution. However, on vector supercomputers, page fault on a vector-mode execution would disturb vector pipelining. It is also reported that large-scale numerical computing has little locality of reference, and simple LRU paging algorithms sometimes tragically declines the performance of it [ASP82]. In fact, few

TABLE V.1. Specifications of extended storage on major vector supercomputers.

Model *Name*	*ES capacity* (max) *Data-transfer speed*	*MM capacity* (max)	*Transfer mode* *Software support*
HITAC S–3800/480	$16GB$	$2048MB$	synchronous/asynchronous
Extended Storage	$4GB/sec$		FORTRAN READ/WRITE
HITAC S–820/80	$12GB$	$512MB$	*do.*
do.	$2GB/sec$		*do.*
NEC SX–3/44	$16GB$	$2048MB$	synchronous (by SPU)
Extended Memory Unit	$3GB/sec$		*do.*
(XMU)			
FACOM VP 2600/20	$8GB$	$2048MB$	*do.*
System Storage	(not announced)		SSU array, READ/WRITE
CRAY Y–MP8/8	$4GB$	$1024MB$	*do.*
Solid-State Disk (SSD)	$2.5GB/sec$		SSD array, READ/WRITE
FACOM VP 400E	$0.67GB$	$256MB$	direct (by VPU)
Vector Storage	$10GB/sec$		allocated by compilers
IBM ES/3090VF	$2GB$	$512MB$	synchronous (by SPU)
Expanded Storage (ES)	$0.1GB/sec$		virtual memory

SPU : Scalar Processing Unit

VPU : Vector Processing Unit

vector supercomputers utilizes virtual memory system based on demand paging.

On the other hand, one can achieve almost the same performance as when all data are on main memory, by designing optimal vector algorithms which requires minimum data transfers between ES and MM. It is shown that external LU factorization and external multidimensional FFT can be performed in *computation bound*, i.e., data-transfer cost is negligible compared with computation cost, for huge data which is fifty times as large as the size of main memory [TO87, TS88]. However, designing optimal I/O algorithms with which arithmetic operations are still fully vectorized is not so easy for ordinary users. Even if a user has a optimized program which works well on main memory, he has to determine the control of data transfer between ES and MM carefully, and has to rewrite array references of the program and insert READ/WRITE statements appropriately into it.

3.2. Virtualization of Extended Storage. As described in Section 2, extended storage has intrinsic supercomputing power in both of the size and the speed, it is not so easy for ordinary users to enjoy it fully. The method proposed here is a solution of this issue [Tsu88b, OOT90]. Our target is a program written without considering the presence of memory hierarchy. When array data of a program becomes larger than the size of available MM, the compiler automatically transforms it so that it does appropriate I/O between MM and ES. The transformation is done without changing the order of arithmetic operations in the original program. As far as array data is concerned, users can write programs as if MM were as large as ES. We may say that ES is virtualized as extension of MM. Needless to say, the compile time transformation affects nothing in execution of other programs. It needs neither dynamic address conversion mechanism nor operating system support, while the both of them are indispensable but cause some overhead in conventional virtual storage systems.

The compiler with the transformation facility automatically detects declarations

of huge arrays that overflow the MM, and converts them into files on ES. Next it extracts control flows of (possibly nested) DO loops, IF-THEN-ELSE structure, and IF-GOTO loops over statements that refer the array data, and inserts codes for data transfers between ES and temporary area on MM at optimal positions. Compared with conventional virtual memory system, the method has the advantage of managing only array data. Data-transfer cost might be minimized by determining partition of arrays and positions at which the codes for data transfers are inserted according to the access patterns of array data in the original program.

Similar approach is adopted in *SSU arrays* on system storage of Fujitsu VP-2000's and SSD arrays of Cray Y/MP's. SSU arrays are specified in JCLs, and syntactically equal to ordinary arrays in programs. SSU arrays are allocated on ES, and data-transfer instructions to ES are put into the object codes at compile time. The goal of SSU array is the same as ours and the approach of it is analogous to ours so far as it is done by compilers, not by the operating system. The essential difference between the SSU array approach and our proposed method is that an SSU array is stiffly allocated on ES in the standard columnwise allocation, while our method carefully chooses the best partitioning and allocation according to the access patterns (as shown in the next section). Comparison of our proposed method to SSU arrays in actual performance is given in Section 7.

4. Allocation of huge arrays on ES

4.1. Partioning a huge array. Data on ES can be accessed only when they are transferred into main memory. Data transfer must be done by a block. Thus we must first determine how to divide the target array into units suitable to block data transfer. We name the unit of array data transferred at a time *a page*, and the unit of work area on main memory where a page is read *a page frame*.

Let us consider how to divide a two-dimensional array for example. Naive colum-

nwise striped page partitioning, which is used in conventional virtual memory (and also SSU arrays), is effective when the array data is accessed only columnwise, viz., by DO loops whose control variables occur only in the first subscript expression of array references. A whole vector access by an iteration of a DO loop can be done as ordinary vector access on main memory by transferring the page that contains the vector before the iteration. However, pitiable overhead arises in vector accesses orthogonal with the direction of partitioning. Consider an example of columnwise partitioning of a two-dimensional array shown in Figure V.16 (a). Each execution of the loop controlled by `I` from 1 to N requires all pages of array `A`. The loop must be divided into doubly nested loops where the innermost loop can be executed on the available page frames. Thus the whole of the array data is read $N/2$ times in one execution of the outer loop. Even if the `I` loop of the original program is vectorizable, only the innermost loop can be vectorized with shorten vector length in the converted program.

If the array is accessed only rowwise, naive rowwise striped partitioning is effective (Figure V.16 (b)). When each of huge arrays is accessed one-directionally, and compile-time analysis can detect it, then the naive one-directional partitioning along the access direction is the best. In fact, a Fortran preprocessor for the transformation with one-directional partitioning based on our early proposal achieved quite a good performance in some typical programs of numerical computing [OOT90]. But there remain arrays which are accessed multi-directionally.

The alternative solution employed here is tiling partitioning [Wol89]. The target array which is accessed in several directions are partitioned over all dimensions. For example, a two-dimensional array is partitioned like a lattice (See Fig. V.17). When a line of data is accessed by a DO loop along a dimension of a multidimensional huge array, the pages that just contain the vector are transferred into page frames on main memory just before the loop is executed. Consecutive vector

```
      DO 10 J=1,N/2
      DO 10 I=1,N
10       A(J,I) = A(N-J,I)
```

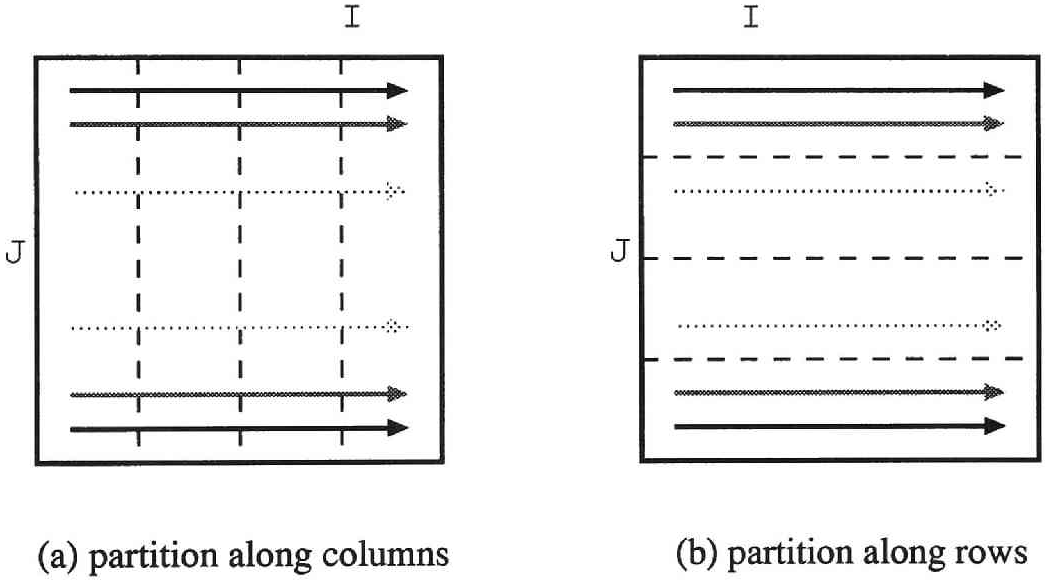


FIGURE V.16. Data-division patterns and amount of data transfer.

accesses over whole of the array requires at most the same number of data transfers as the number of pages, irrespective of the direction of access. Thus tiling partitioning is superior to striped partitioning in uniformity of data-transfer cost on multi-directional accesses.

Next we must determine the page size. The larger the block size of data transfers between MM and ES is, the faster data transfers are performed. However, too large page size may cause more unused data on the transferred pages. If we examined details of the algorithm implemented in the original program, the optimal page size might be determined analytically. But it seems almost impossible to determine the optimal page size for an arbitrarily given program at simple compile-time analysis. So we adopt a compromising page size by which data transfer is performed at least in 50% of the peak speed.

4.2. Vector access to a line of array data. In order to exploit the full performance of vector supercomputers, it is indispensable to keep vector length of vectorized loops enough long. Using tiling partitioning of arrays, it is possible to prepare all data accessed by a loop on main memory before the loop is executed. However, elements of a vector may be allocated in pieces on page frames, although they are to reside at regular intervals in the original program (Fig. V.18). Even if a loop which access the vector is vectorizable in the original program, the corresponding loop in the codes generated by naive transformation can be vectorized only piecewise, with small vector length limited by the side length of pages.

Our proposal to defeat the difficulty is utilization of vector indirect addressing in access of vector data allocated piecewise-linearly on page frames. Preparing a list vector which contains the addresses of indices of the vector data accessed in the target loop, the loop is vectorized in the same vector length as the loop length (Fig. V.18). Most of recent vector supercomputers have special hardware support for fast vector indirect addressing [UT91, UT93].

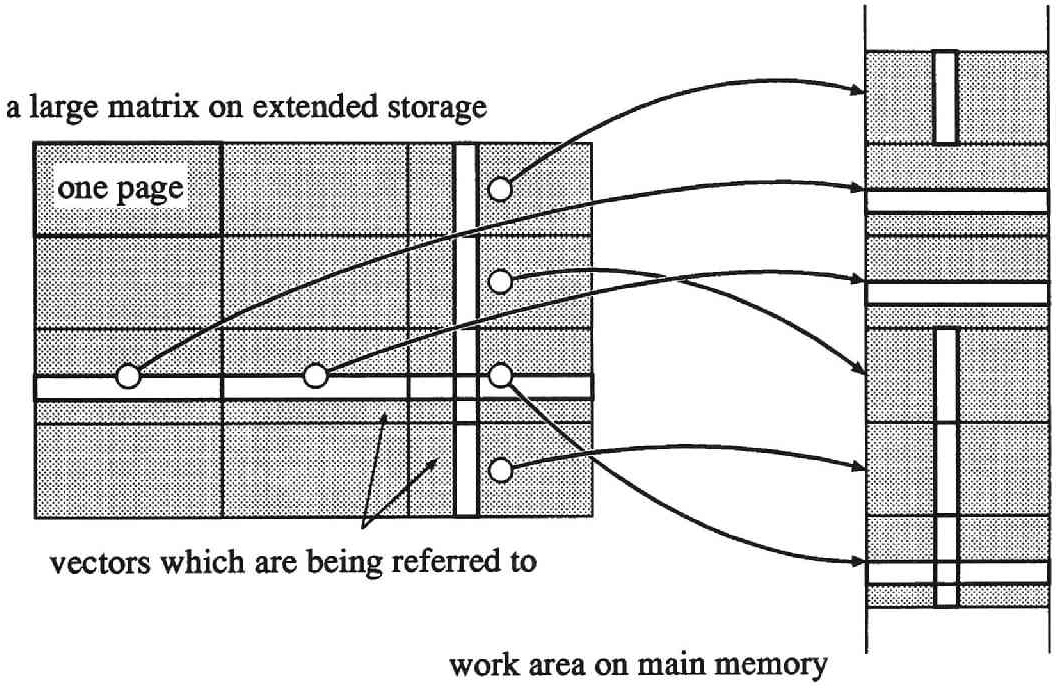


FIGURE V.17. Multi-dimensional partitioning by lattice.

If a vector along a dimension of an array is accessed by a loop, pages that contain the vector is transferred into main memory, and the list vector that gives correspondence of the accessed vector to data on the pages is generated, and then the loop is executed. Of course data transfers are not needed for pages which already reside on main memory. When the array is scanned consecutively, page fetch occurs only at boundaries of pages. Although generation of list vectors is needed at each vector access, the overhead of it is quite little. Details of techniques for generating list vectors is described in [OKT93].

5. Data transfer management

5.1. Page prefetching. A vector load instruction assumes that all data loaded by it are already resides on main memory; otherwise the vector load pipeline might be jammed by page faults. On conventional virtual memory systems based on demand page replacement, it is checked at run time whether the data at the address specified as operands of an instruction is on main memory or not. In contrast, the band of pages that contains the vector accessed in the target loop is transferred beforehand on our proposed system.

Array elements are accessed mostly along only one dimension, i.e., only one of the subscript expression changes, at the innermost loops. The accessed line of data of such an innermost loop is fixed at the point where all subscript expressions except that of the accessed line are fixed. Concerning a two dimensional array, for example, either a row or a column is fixed at some point outside the loop. Then the system inserts codes for transferring pages that contain the accessed line at the point in the source code.

When a huge array is accessed not along a line but elementwise, the page that contains the accessed element is transferred at the point where all subscript expressions of the array access are fixed. Similarly, when two or more of subscript

expressions change at the innermost loop, codes for data transfer are inserted in the loop. See also a proposal of techniques for "oblique" line accesses in Kawabata [Kaw94].

5.2. Page replacement algorithms. Work area on main memory are partitioned into units called *page frames* (Fig. V.18). If an accessed line is specified, the band of pages that contains the line is determined, and pages that are not on work area are transferred from ES into empty page frames. The list vector for indirect addressing is also generated.

If a page that is already referred at another array reference, then the page frame on which the page resides is shared, and only the list vector for the new access is generated. Note that no consistency control of data is needed since a page never resides on two page frames at a time. Each page frame has a reference counter, which keeps the number of active references that share the data on it. The reference counter is incremented when a reference is activated, and is decremented when a reference is inactivated.

When a page becomes not referred at any active reference (viz., the reference counter becomes zero), the page frame is queued to the unreferred page frame list. Note that data on the page frame still remain and are not written back then. If a page on a page frame queued to the list becomes referred again, the page frame is taken off from the list and reused (no data transfer is needed).

When a new page is transferred from ES, the least recently referred page, i.e., the page frame at the top of the unreferred page frame list is chosen. Here pseudo LRU page replacement is implemented. The difference of ours from the LRU page replacement algorithm used in conventional virtual memory system is that a page is released explicitly by a code inserted in the transformation. If the page frame has been modified, the data are written back to the ES; otherwise the data are junked.

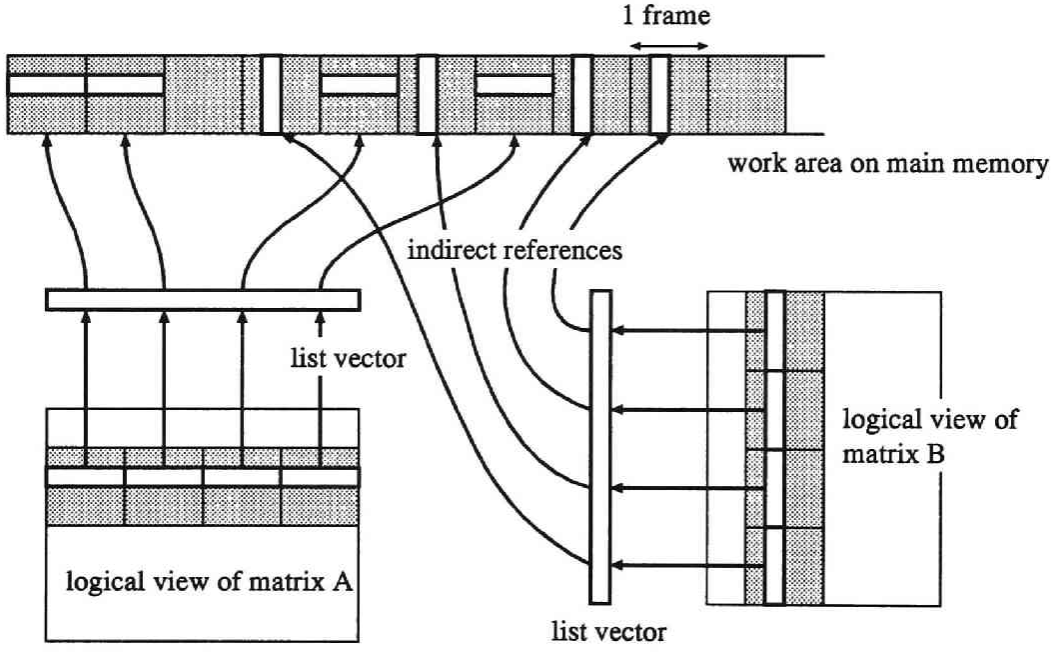


FIGURE V.18. Page frames on work area of main memory.

6. Transformation at source-code level

The proposed transformation can be implemented either as a preprocessor at source-code level, or a extended function of an automatic vectorizing compiler. We have developed a FORTRAN preprocessor as a prototype. The input of our preprocessor is any Fortran program which uses huge arrays but is written without awareness of memory hierarchy. The output is a Fortran program, where huge arrays are allocated as files on ES (by OPEN statements), and data transfers between MM and ES are done by Fortran standard I/Os (READ/WRITE statements). The transformed codes are assumed to be compiled by automatic vectorizing FORTRAN compilers provided by supercomputer manufacturers.

Users specify arrays that are to be allocated on ES and the size of work area by directives in source codes as comment statements. The transformation by the preprocessors are done as follows.

(1) Interprocedure analysis is done in order to detect huge arrays that passed to subroutines as call-by-reference parameters.
(2) Declaration of the array for work area is inserted in the main routine.
(3) The declaration statement of each huge array (DIMENSION statement etc.) is removed. Instead, a CALL statement of a routine creating the array on ES is inserted at the top of executed statements of the module.
(4) Each occurrence of reference of a huge array is numbered its reference number, and is replaced with a reference to work area via a corresponding list vector. The number is choosed so that any two distinct huge array references are numbered the same if and only if they are in the same loop structure and have the same subscripts (with respect to the referred line).
(5) For each huge array reference distinguished by the reference number, a call of the data transfer subroutines and actual parameters for it are inserted.

The insertion point is determined by the control flow and the data flow of the original program, as is stated in the previous section.

Since the transformation is done at source-code level, the power of existing vectorizing compilers can fully be utilized. Also users can edit and optimize the transformed codes manually.

Data transfer control is done by the following four subroutines:

- Initializing the work area.
- Creating an array on ES.
- Making a huge array reference active, or change the reference element/line of an active reference.
- Making a huge array reference inactive.

The routine for initialization is called just once at each execution of a program. It initializes tables for work area management, e.g., unreferred page frame list flags indicating each frame has data or not, etc.

The routine for creating a huge array on ES is called once for each huge-array declaration. It opens a file on ES accessed via the direct access method with a fixed record length. A huge array allocated on ES is identified by its *array number*, which is also used as the device number of the file.

The routine for making a reference active or moving an active reference is invoked at each distinct occurrence of array reference. Here the array number, the reference number, the values of array subscripts, the direction of reference vector, and a flag whether the referred data is defined or not, are passed as parameters. The routine examines whether each of pages referred by the reference vector is on work area or not, and reads it from ES if not. If the reference is a line access, pages for the vector of elements are prepared together, and the list vector for it is also generated or updated.

The routine for making a vector reference inactive is called at the end of a

series of references numbered the same. The page frames which have been used are released, i.e., the link counts of them are decremented, and page frames whose link counts become zero are queued to the unreferred page frame list.

An example of the transformation done by our preprocessor is shown in Fig. V.19. Arrays referred at line 10 of the code in (a) of Fig. V.19 is allocated on ES and the array references in (a) are replaced with references of work area on main memory, i.e., a one-dimensional array `XXTT` in lines 19 and 20 of (b). The first subscript of the array reference of `CX` and the second subscript of that of `VY` in line 10 of (a), viz., `K`, are fixed at line 7 of (a), the data-transfer routines for them are called immediately after the DO statement (lines 12 and 13 of (b)).

7. Performance evaluation

To confirm the effectiveness of our proposed method, we have done a timing measurement on Hitachi's S–3800/480 at the Computer Centre of the University of Tokyo, and on Fujitsu's VP 2600 at Data Processing Center of Kyoto University. We adopted a program of typical LU factorization of dense matrix as a example [Tsu88c]. The program is based on Crout's elimination scheme; rowwise and columnwise eliminations are alternatively done. In order to utilize the arithmetic pipelines fully, two pivoted rows or columns are simultaneously eliminated. The program is originally developed and tuned up to achieve the best performance on Fujitsu VP 200 for matrices with two hundred to five hundred unknowns [Wu87].

A result of timing measurement of the program for LU factorizing 3000 × 3000 matrices, on HITAC S–3800/480 using Hitachi's vectorizing compiler FORTRAN77/HAP is shown in Table V.2. For comparison, three different page sizes, 201×201 ($323KB$), 301×301 ($725KB$) and 401×401 ($1.3MB$) are adopted. We also measured the CPU time when the original program for a 3000×3000 matrix is executed ordinary (not using ES). The results of measurement are summarized

```
      SUBROUTINE KERN21(N,LDA,PX,VY,CX)
      PARAMETER(N0=1000,LOOP=40000)
      INTEGER N, LDA
      REAL*8 PX(LDA,N), VY(LDA,N), CX(LDA,N)
C
      DO 21 L=1,LOOP
      DO 21 K=1,N0
      DO 21 I=1,N0
      DO 21 J=1,N
        PX(I,J)=PX(I,J)+VY(I,K)*CX(K,J)                          10
21    CONTINUE
      RETURN
      END
```

(a) The original loop.

```
      SUBROUTINE KERN21(N,LDA,PX,VY,CX)
      PARAMETER(N0=1000,LOOP=40000)
*     (declarations of constants)
      INTEGER XXII1,XXII2,XXII3
      INTEGER XXI(XXIDMX,XXIRMX), XXDIFF(XXIRMX)
      REAL*8 XXTT(XXIFLN*XXIFMX)
      COMMON /XXT/ XXTT /XXI/ XXI /XXD/ XXDIFF
      INTEGER CX, VY, PX
      INTEGER N, LDA
C                                                                10
      DO 21 L=1,LOOP
      DO 21 K=1,N0
      CALL XXCL(VY,2,1,K,.FALSE.,1,(1),(N0))
      CALL XXCL(CX,3,K,1,.FALSE.,2,(1),(N))
      DO 21 I=1,N0
      CALL XXCL(PX,1,I,1,.TRUE.,2,(1),(N))
*VOPTION INDEP(XXTT), VIST
      DO 21 J=1,N
        XXTT(XXI(J,1)+XXDIFF(1))=XXTT(XXI(J,1)+XXDIFF(1))
     &+XXTT(XXI(I,2)+XXDIFF(2))*XXTT(XXI(J,3)+XXDIFF(3))          20
21    CONTINUE
      CALL XXEX(1)
      CALL XXEX(2)
      CALL XXEX(3)
      RETURN
      END
```

(b) The transformed code.

FIGURE V.19. An example of source-level transformation.

in Table V.3.

The transformed code using work area whose size is about half of the array achieves about 50% of the execution speed when the original one is executed normally on main memory. Even when the size of work area is about one fifth of the array, about 30% of the speed is achieved. The speed of data transfer is almost determined by the page size (about $3.2GB/sec$ when the page size is $1.3MB$). However, the larger the page size is, the more unused data are transferred; the speed of data transfer and the total amount of transferred data are in trade-off.

Next we show timing measurement of the program on Fujitsu's VP 2600. The size of matrices is 3000, and the page size is 301×301. Speed of execution on SSU array, and that on only main memory are also measured.

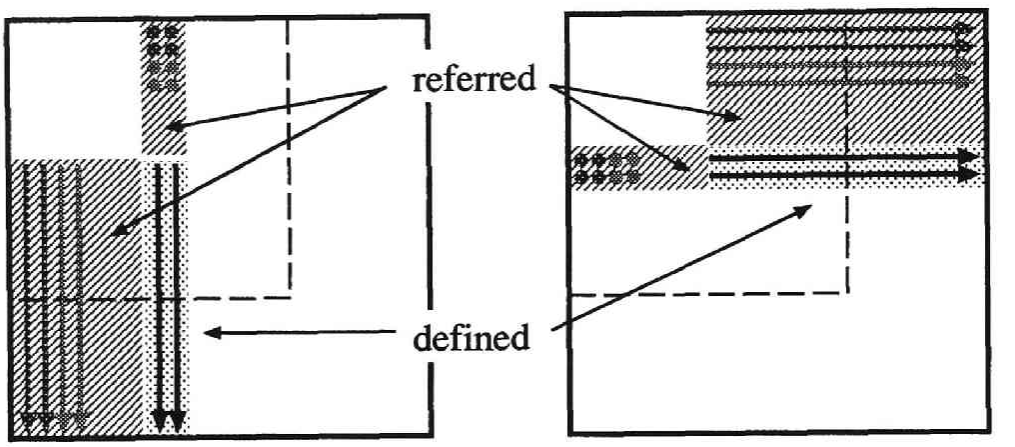
The speed on SSU array is quite lower than that of our proposed method. This seems because rowwise accesses of the array occur so frequently in the program. While data transfer for vector access of contiguous data on SSU array is done in full speed, data transfer for access of data at large intervals is done in quite slow speed [Nag92]. In contrast, the cost of data transfer is uniform in access of any direction in our proposed method. Our method also has advantage in data transfer cost since pages are cached on work area.

8. Considerations

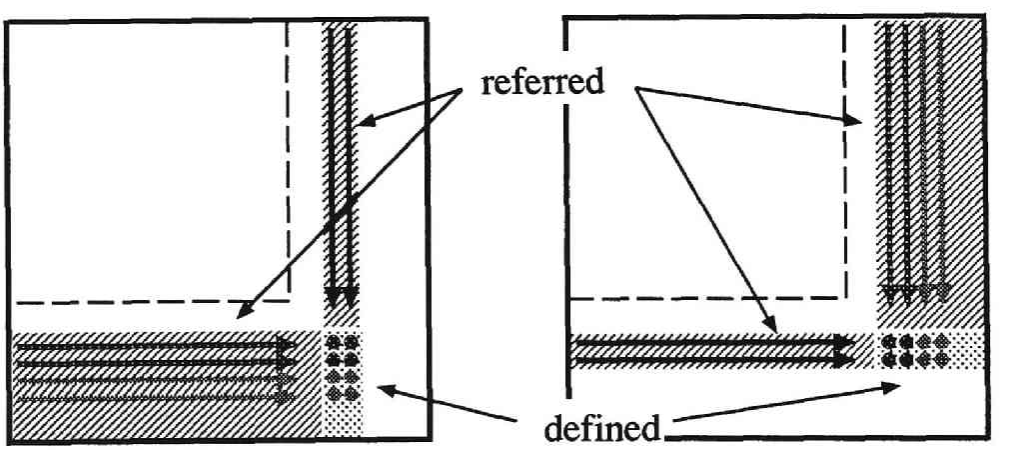
In this chapter, a new method of using extended storage of vector supercomputers as extended main memory is proposed.

We have developed a preprocessor which realize the proposed program conversion and have evaluated the effectiveness of our method by timing measurement. The timing result suggests that the proposed method can be applied to various range of programs with enough efficiency.

Our previous works [TO87, TS88] suggest that it seems required to change the



(a) Access patterns in the formar half.



(b) Access patterns in the latter half.

FIGURE V.20. Data reference patterns of the LU factorization.

control flows and the orders of operations in order to reduce the cost of data transfer relatively negligible in comparison with the cost of arithmetic operations. Further research is required to automate such powerful program conversion that can change the algorithm of the program without changing the semantics of it.

TABLE V.2. Performance of LU factorization on HITAC S–3800.

Size of matrices: 3000 × 3000

Block size	**MM usage**		**CPU time**			**Data transfer**		**Performance**	
MB	*MB*		*sec*	VPU	Data transfer	Speed *GB/sec*	Amount *GB*	*Mflops*	
on main memory	72.0	(100%)	8.0	6.9	—	—	—	2250	(100%)
201 × 201	14.5	(20%)	27.6	11.7	14.3	2.57	36.8	653	(29%)
(323*KB*)	19.4	(27%)	24.4	11.7	11.1	2.52	28.0	739	(33%)
	24.2	(34%)	22.2	11.7	8.7	2.54	22.2	811	(36%)
	29.1	(40%)	20.1	11.7	6.4	2.54	16.1	896	(40%)
	33.9	(47%)	18.3	11.7	4.4	2.53	11.1	986	(43%)
301 × 301	21.7	(30%)	23.9	11.2	10.7	3.00	32.1	753	(33%)
(725*KB*)	29.0	(40%)	19,9	11.2	6.5	3.00	19.5	905	(40%)
	36.2	(50%)	17.4	11.2	3.9	3.00	1.3	1035	(46%)
401 × 401	30.9	(43%)	23.4	11.1	10.7	3.20	34.3	769	(35%)
(1286*KB*)	41.2	(57%)	18.7	11.1	5.3	3.20	16.9	962	(43%)
	51.5	(72%)	15.6	11.1	2.3	3.19	7.2	1152	(51%)

Measured on S–3800/480 at Computer Centre of the University of Tokyo in Oct. 1993.

TABLE V.3. Performance of LU factorization on FACOM VP 2600.

Size of matrices: 3000 × 3000

Block size	**Usage of main memory**	**CPU time**		**Performance**
MB	*MB*	*sec*	VPU	*Mflops*
(on main memory)	72.0 (100%)	16.3	16.3	1101 (100%)
our method	29.0 (40%)	54.7	19.5	329 (30%)
SSU array	—	314.0	24.9	57 (5%)

Measured on VP 2600 at Data Processing Center of Kyoto University in Apr. 1993.

CHAPTER VI
Conclusion

Four topics on theoretical and fundamental aspects of supercomputing softwares are discussed.

The topics in Chapter II and Chapter III are discussed on hardware models with extreme parallelism, i.e., "How much speed up is possible if any number of processors are available?" The results shown in these chapters indicate the theoretical potentialities and limits of massively parallel systems processing logical programs. The results are also useful for analysis of parallel algorithms written in logic programming languages. Moreover, they might have possibility to realize the ultimate supercomputing environment such that an extremely parallel hardware algorithm written in a logic program is processed by, what is called, a silicon compiler, and is hardwired into a circuit which works with the algorithm in the theoretically maximum speed.

In Chapter III, the class of queries in $\mathcal{P}$ and the class of queries in $\mathcal{NC}$ are characterized and classified via logical query programs. While arbitrary logic programs have as much power as that of Turing machines, the class of logical queries is equal to $\mathcal{P}$, and hence subclasses of queries which are effectively parallelizable (viz., in $\mathcal{NC}$), such as the class of queries defined by linear programs, or that by non-confluent programs, can naturally be defined with some syntactic restrictions.

The presented procedure of program transformation gives additional rules for effectively parallelized derivation to a logical query program which have potential $\mathcal{NC}$-parallelism in it but itself works serially (in polynomial time). It might be a basis of future research on parallelizing supercompiler which extract as much parallelism as there resides from a logic program written by an ordinary user without explicit awareness of parallelism.

In Chapter IV, we have argued on the lower bounds of data transfer complexity and optimal algorithms which perform the bounds for sorting, permutation, FFT, etc., on bus-connected parallel two-level memories. The difference between models with broadcasting and those without broadcasting has been clearly stood out. The complexity on models with network topologies other than a shared bus, and the complexity of bus-connected parallel two-level memory model where each processor has a local disk, are remained as open problems.

In Chapter V, virtualization of secondary high-speed extended storages of vector supercomputers as extended main memory is discussed. The proposed program transformation makes it possible to execute an ordinary program on extended storage as fast as when it is executed on main memory. Vectorizability of the program is preserved by the transformation by utilizing lattice-wise partitioning of huge arrays into pages, page prefetching before vector-mode instructions, and indirect vector load/store of pages randomly scattered on the temporary area. Further researches are desired to implement flexible and dynamic change of partitioning and addressing of array data, and thoroughgoing minimization of data transfers by global instruction reordering such as loop unrolling and/or loop interchanging.

Bibliography

[Aki85] Selim G. Akil, *Parallel sorting algorithms*, Academic Press, London, 1985.

[AP93] Foto Afrati and Christos H. Papadimitriou, *The parallel complexity of simple logic programs*, J. ACM **40** (1993), no. 4, 891–916.

[ASP82] W. A. Abu-Sufah and D. Padua, *Some results on the working set anomalies in numerical programs*, IEEE Trans. Software Eng. **SE-8** (1982), no. 2, 97–106.

[AU79] A. V. Aho and J. D. Ullman, *Universality of data retrieval languages*, 6th Sympos. on Principles of Programming Lagnuages, 1979, pp. 110–117.

[AV88] A. Aggarwal and J. S. Vitter, *The input/output complexity of sorting and related problems*, Comm. ACM (1988), 1116–1127.

[BCD+87] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa, *Two applications of complementation via inductive counting*, Tech. report, IBM T. J. Watson Research Center, Hampton, VA, 1987.

[BFP+73] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, *Time bounds for selection*, J. Comput. Syst. Sci. **7** (1973), 448–461.

[CH84] A. Chandra and D. Harel, *Structure and complexity of relational queries*, J. Comput. System Sci. **25** (1984), 99–128.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer, *Alternation*, J. ACM **28** (1981), no. 1, 114–133.

[Cod70] E. F. Codd, *A relational model for large shared data banks*, Comm. ACM **13** (1970), no. 6, 377–387.

[Coo85] Stephan A. Cook, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control **64** (1985), 2–22.

[Fer86] S. Fernbach (ed.), *SUPERCOMPUTERS class IV systems, hardware and software*, North-Holland, Amsterdam, 1986.

[Flo72] R. W. Floyd, *Permuting information in idealized two-level storage*, Complexity of Computer Calculations (R. Miller and J. Thatcher, eds.), Plenum, New York, 1972, pp. 105–109.

[FW78] S. Fortune and J. Wyllie, *Parallelism in random access machines*, Proc. 10th ACM Symp. on Theory of Computing, 1978, pp. 114–118.

[GMS93] Haim Gaifman, Harry Mairson, and Yehoshua Sagiv, *Undecidable optimazation problems for database logic prgrams*, J. ACM **40** (1993), no. 3, 683–713.

[Gre73] Sheila A. Greibach, *The hardest context-free language*, SIAM J. Comput. **2** (1973), no. 4, 304–310.

[HJ88a] R. W. Hockney and C. R. Jesshope, *Parallel computer 2: architecture, programming and algorithms*, Adam Hilger, Bristol and Philadelphia, 1988.

[HJ88b] R. W. Hockney and C. R. Jesshope, Parallel Computer 2: architecture, programming and algorithms, ch. 3, 3.3 Switching Networks, pp. 259–285, In Hockney and Jesshope [HJ88a], 1988.

[HK81] J. W. Hong and H. T. Kung, *I/O complexity: the red-bule pebble game*, Proc. of the 13th Annual ACM Symposium of Theory of Computing, 1981, pp. 326–333.

[Imm86] Neil Immerman, *Relational queries computable in polynomial time*, Inform. and Control **68** (1986), 86–104.

[Imm87] Neil Immerman, *Langages that capture complexity classes*, SIAM J. Comput. **16** (1987), no. 4, 761–778.

[Imm88] Neil Immerman, *Nondeterministic space is closed under complementation*, SIAM J. Comput. **17** (1988), no. 5, 935–938.

[Iwa90] Kazuo Iwama, *Extremely parallel algorithms*, J. IPSJ **31** (1990), no. 7, 913–920 (Japanese).

[Iwa92a] Kazuo Iwama, *Parallel algorithms over PRAMs*, J. IPSJ **33** (1992), no. 9, 1033–1041 (Japanese).

[Iwa92b] Kazuo Iwama, *Theory of parallel computation*, J. IEICE **75** (1992), no. 1, 56–65 (Japanese).

[KaHMK+88] S. Kawabe, F. Kobayashi adn H. Murayama, Y. Kiryuu, H. Handa, F. Tagami, S. Gotoo, and T. Aoyama, *The single processor S-820: Peak speed 2GFLOPS*, Nikkei Electronics **437** (1988), 111–125 (Japanese).

[Kan88] Paris C. Kanellakis, *Logic programming and parallel complexity*, In Minker [Min88], pp. 547–585.

[Kas87a] Takumi Kasai, 計算量の理論, ch. 6, 6.6 対数領域と多項式時間, pp. 160–163, In Kasai [Kas87c], 1987.

[Kas87b] Takumi Kasai, 計算量の理論, ch. 5, 5.3 文脈自由言語, pp. 114–115, In Kasai [Kas87c], 1987.

[Kas87c] Takumi Kasai, 計算量の理論, コンピュータサイエンス大学講座, vol. 17, Kindai Kagaku Sha, 1987 (Japanese), ISBN 4–7649–0130–7.

[Kaw94] Hideyuki Kawabata, *Virtualization of semiconductor extended storage as extended main storage for vector supercomputers*, Master's thesis, Department of Information Sci., Kyoto University, 1994.

[Knu73] D. E. Knuth, Sorting and Searching, The Art of Computer Programming, vol. 3, 73, The Art of Computer Programming, Addison-Wesley, 1973.

[Mat90] Toshiko Matsuda (ed.), 京都大学パートナーシッププログラム *1990* 年度研究報告書, 日本 IBM, 1990.

[Min88] J. Minker (ed.), *Foundation of deductive databases and logic programming*, Morgan Kaufmann, Los Altos, 1988.

[Miy91] Satoru Miyano, *Complexity theory for parallel algorithms*, J. IPSJ **32** (1991), no. 2, 171–179 (Japanese).

[Nag92] Toru Nagai, ベクトル計算機入門 *(3):SSU*, Nagoya Univ. Computation Center News **23** (1992), no. 3, 202–213 (Japanese).

[Nau89] Jeffrey F. Naughton, *Minimizing function-free recursive inference rules*, J. ACM **13** (1989), no. 1, 69–91.

[OKT93] Yasuo Okabe, Hideyuki Kawabata, and Takao Tsuda, *Virtualization of semiconductor extended storage as extended main storage for vector supercomputers*, submitted to Trans. IPSJ, 1993.

[ONK86] T. Okada, S. Nagashima, and S. Kawabe, *Hitachi supercomputer S-810 array processor system*, In Fernbach [Fer86], pp. 113–136.

[OOT90] Yasuo Okabe, Shigeyuki Ohnishi, and Takao Tsuda, *Virtualization of semiconductor extended storage as extended main storage for large-scale supercomputing*, Record of the 1990 Kansai-Section Joint Convention of IEEJ, no. S9-2, Nov. 1990 (Japanese).

[OT90a] Yasuo Okabe and Nobutaka Takahashi, NIC における Expanded Storage の効果的利用法, 79–93, In Matsuda [Mat90], 1990 (Japanese).

[OT90b] Yasuo Okabe and Takao Tsuda, 拡張記憶の効率的利用のためのプログラム変換, 73–77, In Matsuda [Mat90], 1990 (Japanese).

[Ruz80] Walter L. Ruzzo, *Tree-size bounded alternation*, Journal of Computer and System Sciences **21** (1980), no. 2, 218–235.

[Ruz81] Walter L. Ruzzo, *On uniform circuit complexity*, Journal of Computer and System Sciences **22** (1981), no. 3, 365–383.

[Sag88] Yehoshua Sagiv, *Optimizing Datalog programs*, In Minker [Min88], pp. 659–698.

[Sha49] C. E. Shannon, *The synthesis of two terminal switching circuits*, BSTJ **28** (1949), 59–98.

[Sha83] Ehud Y. Shapiro, *A subset of Concurrent Prolog and its interpreter*, Tech. Report TR-003, ICOT, 1983.

[Sha84] Ehud Y. Shapiro, *Alternation and the computational complexity of logic programs*, J. Logic Programming **1** (1984), 19–33.

[ŠŠ86] Petr Štěpánek and Olga Štěpánková, *Logic programs and alternation*, 3rd International Conference on Logic Programming (Ehud Y. Shapiro, ed.), Lecuture Notes in Computer Science, vol. 255, Springer Verlag, 1986, pp. 99–106.

[Str69] V. Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik **13** (1969), 354–356.

[SV84] L. Stockmeyer and U. Vishkin, *Simulation of parallel random access machines by circuits*, SIAM J. Comput. **13** (1984), no. 2, 409–422.

[SV85] J. Savage and J. S. Vitter, *Parallelism in space-time tradeoffs*, VLSI: Algorithms and Architectures (P. Bertolazzi and F. Luccio, eds.), Elsevier Science Publishers B. V., 1985, pp. 49–58.

[Sze87] R. Szelepcsényi, *The method of forcing for nondeterministic automata*, Bulletin of EATCS **33** (1987), 96–100.

[TO87] Takao Tsuda and Yasuo Okabe, *Use of semiconductor extended storage as extended main storage for large-scale supercomputing*, Proc. of the Second International Conference on Supercomputing, vol. 1, International Supercomputing Institute, May 1987, pp. 167–183.

[Tom86a] Shinji Tomita, *Parallel computer architectures*, Shōkōdō, 1986 (Japanese), ISBN 4–7856–3066–3.

[Tom86b] Shinji Tomita, Parallel Computer Architectures, ch. 3, 3.3 多段結合網, pp. 80–99, In Tomita [Tom86a], 1986.

[TS88] Takao Tsuda and Yoshiki Seo, *Supercomputing external multidimensional FFT: Use of semiconductor extended storage as extended main storage*, J. Information Processing **11** (1988), no. 2, 112–119.

[TST83] Takao Tsuda, Tatoshi Sato, and Takaaki Tatsumi, *Generalization of Floyd's model on permuting information in idealized two-level storage*, Inf. Process. Lett. **16** (1983), no. 4, 183–188.

[Tsu88a] Takao Tsuda, 数値処理プログラミング, The IWANAMI Software Science Series, vol. 9, Iwanami Shoten, 1988 (Japanese), ISBN 4–00–010349–0.

[Tsu88b] Takao Tsuda, 数値処理プログラミング, ch. 3, 3.7 大規模 LU 分解ベクトルプログラム, p. 148, In Tsuda [Tsu88a], 1988.

[Tsu88c] Takao Tsuda, 数値処理プログラミング, ch. 3, 3.5 LU 分解ベクトルプログラム, pp. 126–130, In Tsuda [Tsu88a], 1988.

[Ued85] K. Ueda, *Guarded Horn Clauses*, Proc. Logic Programming Conference '85, 1985.

[Ume90] Hiroshi Umeo, *Recent studies on parallel algorithms for mesh-connected computers with broadcast buses*, J. IPSJ **31** (1990), no. 7, 906–912 (Japanese).

[Ume91] Hiroshi Umeo, 超並列計算機アーキテクチャとそのアルゴリズム, Kyōritsu Shuppan, 1991 (Japanese), ISBN 4–320–02568–7.

[UT91] Tetsutaro Uehara and Takao Tsuda, *Benchmarking vector indirect load/store instructions*, SUPERCOMPUTER **46** (1991), 15–29.

[UT93] Tetsutaro Uehara and Takao Tsuda, *Benchmarking vector indirect load/store*

instructions, Proceedings of Workshop on Benchmarking and Performance Evaluation in High Performance Computing, July 1993, pp. 16–25.

[UV88] Jeffrey D. Ullman and Allen Van Gelder, *Parallel complexity of logical query programs*, Algorithmica **3** (1988), 5–42.

[VS90] J. S. Vitter and E. A. M. Shriver, *Algorithms for parallel memory I: Two-level memories*, Tech. Report CS-90-21, Department of Computer Science, Brown University, Sept. 1990.

[WKI86] T. Watanabe, H. Katayama, and A. Iwaya, *Introduction of NEC supercomputer SX system*, In Fernbach [Fer86], pp. 153–167.

[Wol89] Michael Wolfe, Optimizing Supercompilers for Supercomputers, vol. 1, 119–123, MIT Press, Cambridge, Mass., 1989.

[Wu87] 呉 永化, *VP* 向き二列同時消去 *LU* 分解法, Proc. of the 2nd Symposium on Applications of Vector Supercomputers, Data Processing Center, Kyoto Univ., Mar. 1987, pp. 8–15 (Japanese).

[Yas92] Hiroto Yasuura, *Parallel computers and parallel computation models*, J. IPSJ **33** (1992), no. 9, 1024–1032 (Japanese).

Acknowledgements

I would like to express sincere gratitude to Professor Takao Tsuda of Kyoto University. He gave me the opportunity of this research, and has been giving me continuous guidance, interesting suggestions, intensive criticism and encouragements during this research.

I would also like to appreciate Professor Shuzo Yajima of Kyoto University who introduced me the research field of computational complexity theory and has been giving me invaluable suggestions and encouragements.

I am indebt to Associate Professor Yoshitoshi Kunieda of Kyoto University, not only for his continuous guidance and accurate criticism during this research, but for reading the early drafts of this thesis carefully and suggesting many illuminating addition and improvements.

I also acknowledge the interesting comments that I have received from Professor Eiji Okubo of Ritsumeikan University, Professor Hiroto Yasuura of Kyushu University, and Associate Professor Naofumi Takagi of Kyoto University.

I would also like to thank Mr. Hideyuki Kawabata who implemented the FORTRAN preprocessor proposed in Chapter V.

Thanks are also due to all members of Professor Tsuda's Laboratory for their discussions and supports throughout this research.

List of Publications by the Author

Major publications.

[1] Takao Tsuda and Yasuo Okabe, *Use of semiconductor extended storage as extended main storage for large-scale supercomputing*, Proceeding of the Second International Conference on Supercomputing (May 1987), pp. 176–183.

[2] Yasuo Okabe and Shuzo Yajima, *Parallel computational complexity of logic programs and alternating Turing machines*, Proceedings of the International Conference on Fifth Generation Computer Systems 1988 (Nov. 1988), pp. 356–363.

[3] Yasuo Okabe and Shuzo Yajima, *Parallel complexity of logic programs and highly parallel computation by logic circuits*, Trans. of IPSJ, **31**, 6 (June 1990), pp. 840–848 (Japanese).

[4] Yasuo Okabe, Naofumi Takagi and Shuzo Yajima, *Log depth circuits for elementary functions using residue number system*, Trans. of IEICE, **J73–D–I**, 9 (Sept. 1990), pp. 723–728 (Japanese); English translation is in ELECTRONICS and COMMUNICATIONS in JAPAN (Scripta Technica, Inc.), Part III: Fundamental Electronics Science, **74**, 8 (Aug. 1991), pp. 31–38.

[5] Yasuo Okabe and Takao Tsuda, *Optimal data-transfer algorithms on bus-connected parallel machines*, submitted to Trans. IPSJ (Japanese).

[6] Yasuo Okabe, Hideyuki Kawabata and Takao Tsuda, *Virtualization of semiconductor extended storage as extended main storage for vector supercomputers*, submitted to Trans. IPSJ (Japanese).

Technical reports.

[1] Yasuo Okabe, Naofumi Takagi and Shuzo Yajima, $O(\log n)$ *depth n-bit binary dividers using residue number system*, Report of Tech. Group on Automata and Languages, IECEJ, AL85–89 (Mar. 1986), pp. 35–44 (Japanese).

[2] Naofumi Takagi, Yasuo Okabe, Hiroto Yasuura and Shuzo Yajima, *Modulo-m addition using redundant representation and its application to residue number / binary conversion*, Report of Tech. Group on Computation, IECEJ, COMP86–14 (June 1986), pp. 39–44 (Japanese).

[3] Takao Tsuda and Yasuo Okabe, *Use of semiconductor extended storage as extended main storage II: solving a system of linear equations that has a dense coefficient matrix with about 10000 unknowns*, Proc. of the 3rd Symposium on Applications of Vector Supercomputers (Data Processing Center, Kyoto Univ.; Mar. 1987), pp. 8–15.

[4] Yasuo Okabe, Naofumi Takagi and Shuzo Yajima, *Log depth circuits for elementary functions using residue number system*, LA Symposium 1987 Summer (July 1987) (Japanese).

[5] Yasuo Okabe and Shuzo Yajima, *Parallel computational complexity of logic programs and alternating Turing machines*, Report of Tech. Group on Computation, IEICE, COMP87–50 (Nov. 1987), pp. 51–60 (Japanese).

[6] Yasuo Okabe and Shuzo Yajima, *Area-time efficient evaluation of elementary functions*, LA Symposium 1988 Winter (Feb. 1988), in Papers Res.

Inst. Math. Sci., Kyoto Univ., **666** 「計算機アルゴリズムの基礎理論」 (July 1988), pp. 115–124.

[7] Yasuo Okabe and Takao Tsuda, *Logspace bounded alternation and logical query programs*, LA Symposium 1992 Winter (Feb. 1992), in Papers Res. Inst. Math. Sci., Kyoto Univ., **790** 「理論計算機科学とその周辺」 (July 1992), pp. 269–275.

[8] Yasuo Okabe and Takao Tsuda, *Optimal data-transfer algorithms on bus-connected parallel machines*, LA Symposium 1993 Winter (Feb. 1993), in Papers Res. Inst. Math. Sci., Kyoto Univ., **833** 「計算機構とアルゴリズム」 (Apr. 1993), pp. 250–255 (Japanese).

[9] Yasuo Okabe, Hideyuki Kawabata and Takao Tsuda, *Virtualization of semiconductor extended storage as extended main memory*, Preprents Work. Gr. for High-Performance Computing, IPSJ, 93–HPC–46–9 (Apr. 1993), pp. 53–59 (Japanese).

[10] Yasuo Okabe and Takao Tsuda, *Optimal I/O algorithms on bus-connected parallel machines*, Preprents Work. Gr. for Algorithms, IPSJ, 93–AL–34–5 (Aug. 1993), pp. 33–40 (Japanese).

Convention records.

[1] Yasuo Okabe, Naofumi Takagi and Shuzo Yajima, *Conversion algorithms between residue numbers and binary numbers using redundant representation*, Record of the 1986 Kansai-Section Joint Convention of IEEJ (Nov. 1986), , G6–12 (Japansese).

[2] Yasuo Okabe, Naofumi Takagi and Shuzo Yajima, *Log depth circuit for elementary functions using residue number system*, Proc. 34th Annual Convention IPSJ, 3N–5 (Japanese).

[3] Yasuo Okabe and Shuzo Yajima, *Area-time optimal square rooting circuits for* $T = \Omega(\log^{1+\epsilon} n)$, Papers Annual Meet. on Information and Systems of IEICE (Mar. 1988), D–342 (Japanese).

[4] Shigeyuki Ohnishi, Yasuo Okabe and Takao Tsuda, *Advanced use of semiconductor extended storage as extended main storage for supercomputing*, Proc. 39th Annual Convention IPSJ (Oct. 1989), 1Q–2 (Japanese).

[5] Yasuo Okabe, Shigeyuki Ohnishi and Takao Tsuda, *Virtualization of semiconductor extended storage as extended main storage for large-scale supercomputing*, Record of the 1990 Kansai-Section Joint Convention of IEEJ (Nov. 1990), S9–2 (Japanese).

[6] Yasuo Okabe, Hideyuki Kawabata and Takao Tsuda, *Virtualization of semiconductor extended storage as extended main storage for vector supercomputers*, 9th Conf. Proc. JSSST (Sept. 1992), D6–1 (Japanese).

[7] Ichiro Mizunuma, Tetsutaro Uehara, Yasuo Okabe, Yoshitoshi Kunieda and Takao Tsuda, *Data-dependence analysis of nested loops containing symbolics and nonlinear expressions*, 9th Conf. Proc. JSSST (Sept. 1992), D6–2 (Japanese).

[8] Yasuo Okabe, Sy-ya Wang and Takao Tsuda, *Analysis of algorithms for linear computation on bus-connected vector multiprocessors*, Proc. 45th Annual Convention IPSJ (Oct. 1992), 4L–2 (Japanese).

[9] Yuji Ohnishi, Yasuo Okabe, Eiji Okubo and Takao Tsuda, *A thread distributor in the distributed operating system DM-1*, Proc. 48th Annual Convention IPSJ (Mar. 1994), 2H–8 (Japanese).

[10] Akio Hamada, Yasuo Okabe, Eiji Okubo and Takao Tsuda, *System shutdown for fault tolerance in the distributed operating system DM-1*, Proc. 48th Annual Convention IPSJ (Mar. 1994), 4F–7 (Japanese).